

Quartus II Version 7.2 Handbook

Volume 1: Design and Synthesis

101 Innovation Drive
San Jose, CA 95134
www.altera.com

QII5V1-7.2

Copyright © 2007 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

I.S. EN ISO 9001

Contents

Chapter Revision Dates ..... **xv**

About this Handbook ..... **xvii**
How to Contact Altera xvii
Third-Party Software Product Information xvii
Typographic Conventions xviii

Section I. Design Flows

Chapter 1. Design Planning with the Quartus II Software
Introduction 1–1
Device and Programming/
Configuration Method Selection 1–2
Device Selection 1–2
Device Migration Planning 1–3
Programming/Configuration Method Selection 1–4
Early Planning Tools for Power and I/O 1–5
Early Power Estimation 1–5
Early Pin Planning and I/O Analysis 1–6
Selecting Third-Party EDA Tool Flows 1–9
Synthesis Tools 1–9
Simulation Tools 1–10
Formal Verification Tools 1–10
Planning for On-Chip Debugging Options 1–11
Planning for an Incremental Compilation Flow 1–13
Flat Compilation Flow with No Design Partitions 1–13
Incremental Compilation with Design Partitions 1–14
Top-Down Versus Bottom-Up Incremental Flows 1–15
Planning Design Partitions 1–17
Creating a Design Floorplan 1–18
Early Timing Estimation 1–19
Conclusion 1–20
Referenced Documents 1–20
Document Revision History 1–21

Chapter 2. Quartus II Incremental Compilation for Hierarchical and Team-Based Design
Introduction 2–1
Choosing a Quartus II Compilation Flow 2–3

Flat Compilation Flow with No Design Partitions 2–3
Incremental Compilation Flow with Design Partitions 2–5
Top-Down versus Bottom-Up Compilation Flows 2–9
Quick Start Guide – Summary of Steps for an Incremental Compilation Flow 2–11
Top-Down Incremental Compilation Flow 2–11
Bottom-Up Incremental Compilation 2–13
Design Partitions 2–17
Design Partition Assignments Compared to Physical Placement Assignments 2–18
Creating Design Partitions 2–19
Partition Name 2–21
Setting the Netlist Type for Design Partitions 2–22
Fitter Preservation Level 2–24
Empty Partitions 2–26
What Represents a Source Change for Incremental Compilation? 2–27
Creating a Design Floorplan With LogicLock Location Assignments 2–29
Taking Advantage of the Early Timing Estimator 2–31
Exporting and Importing Partitions for Bottom-Up Design Flows 2–32
Quartus II Exported Partition File (.qxp) 2–32
Exporting a Lower-Level Partition to be Used in a Top-Level Project 2–33
Exporting a Lower-Level Block within a Project 2–35
Importing a Lower-Level Partition Into the Top-Level Project 2–36
Importing Assignments and Advanced Import Settings 2–37
Generating Bottom-Up Design Partition Scripts for Project Management 2–40
Guidelines for Creating Good Design Partitions and LogicLock Regions 2–46
Creating Good Design Partitions 2–47
Partition Statistics Reports 2–50
Resource Balancing 2–51
Timing Budgeting 2–53
Methodology to Check Partition Quality during Partition Planning 2–54
The Importance of Floorplan Location Assignments in Incremental Compilation 2–55
Creating Good Floorplan Location Assignments 2–57
Incremental Compilation Advisor 2–60
Criteria for Successful Partition and Floorplan Schemes 2–61
Recommended Design Flows and Compilation Application Examples 2–62
Top-Down Incremental Design Flows 2–62
Bottom-Up Incremental Design Flows 2–67
Incremental Compilation Restrictions 2–76
Using Incremental Synthesis Only Instead of Full Incremental Compilation 2–76
Preserving Exact Timing Performance 2–77
Using Incremental Compilation with Quartus II Archive Files 2–77
Formal Verification Support 2–78
OpenCore Plus MegaCore Functions in Bottom-Up Flows 2–78
Importing Encrypted IP Cores in Bottom-Up Flows 2–78
SignalProbe Pins and Engineering Change Management with the Chip Planner 2–78
SignalTap II Embedded Logic Analyzer in Bottom-Up Compilation Flows 2–80
Logic Analyzer Interface in Bottom-Up Compilation Flows 2–81
Migrating Projects with Design Partitions to Different Devices 2–81

HardCopy Compilation Flows 2–82
Assignments Made in HDL Source Code in Bottom-Up Flows 2–83
Compilation Time with Physical Synthesis Optimizations 2–83
Restrictions on Megafunction Partitions 2–84
Routing Preservation in Bottom-Up Compilation Flows 2–84
Bottom-Up Design Partition Script Limitations 2–84
Register Packing and Partition Boundaries 2–87
I/O Register Packing 2–87
Scripting Support 2–99
Generate Incremental Compilation Tcl Script Command 2–99
Preparing a Design for Incremental Compilation 2–100
Creating Design Partitions 2–100
Setting Properties of Design Partitions 2–101
Creating Good Floorplan Location Assignments—Excluding or Filtering Certain Device Elements (Such as RAM or DSP Blocks) 2–102
Generating Bottom-Up Design Partition Scripts 2–103
Exporting a Partition to be Used in a Top-Level Project 2–105
Importing a Lower-Level Partition into the Top-Level Project 2–106
Makefiles 2–106
Recommended Design Flows and Compilation Application Examples 2–107
Conclusion 2–109
Referenced Documents 2–109
Document Revision History 2–111

Chapter 3. Quartus II Design Flow for MAX+PLUS II Users

Introduction 3–1
Chapter Overview 3–1
Typical Design Flow 3–2
Device Support 3–3
Quartus II GUI Overview 3–4
Project Navigator 3–4
Node Finder 3–4
Tcl Console 3–4
Messages 3–4
Status 3–5
Setting Up MAX+PLUS II Look and Feel in Quartus II 3–6
MAX+PLUS II Look and Feel 3–7
Compiler Tool 3–9
Analysis and Synthesis 3–10
Partition Merge 3–10
Fitter 3–10
Assembler 3–11
Timing Analyzer 3–11
EDA Netlist Writer 3–11
Design Assistant 3–11
MAX+PLUS II Design Conversion 3–12
Converting an Existing MAX+PLUS II Design 3–12

Converting MAX+PLUS II Graphic Design Files 3–13
Importing MAX+PLUS II Assignments 3–14
Quartus II Design Flow 3–15
Creating a New Project 3–16
Design Entry 3–16
Making Assignments 3–20
Synthesis 3–23
Functional Simulation 3–24
Place and Route 3–26
Timing Analysis 3–27
Timing Closure Floorplan 3–29
Timing Simulation 3–31
Power Estimation 3–33
Programming 3–33
Conclusion 3–34
Quick Menu Reference 3–35
Quartus II Command Reference for MAX+PLUS II Users 3–36
Referenced Documents 3–45
Document Revision History 3–46

Chapter 4. Quartus II Support for HardCopy Series Devices

Introduction 4–1
HardCopy II Device Support 4–1
HardCopy II Design Benefits 4–1
Quartus II Features for HardCopy II Planning 4–2
HardCopy II Development Flow 4–3
Designing the Stratix II FPGA First 4–4
Designing the HardCopy II Device First 4–6
HardCopy II Device Resource Guide 4–8
HardCopy II Companion Device Selection 4–10
HardCopy II Recommended Settings in the Quartus II Software 4–12
Limit DSP and RAM to HardCopy II Device Resources 4–12
Enable Design Assistant to Run During Compile 4–12
Timing Settings 4–13
Constraints for Clock Effect Characteristics 4–15
Quartus II Software Features Supported for HardCopy II Designs 4–17
Performing ECOs with Quartus II Engineering Change Management with the Chip Planner 4–20
Migrating One-to-One Changes 4–20
Migrating Changes that Must be Implemented Differently 4–21
Changes that Cannot be Migrated 4–22
Overall Migration Flow 4–22
Preparing the Revisions 4–22
Applying ECO Changes 4–23
Formal Verification of Stratix II and HardCopy II Revisions 4–24
HardCopy II Utilities Menu 4–25
Companion Revisions 4–26
Compiling the HardCopy II Companion Revision 4–28

Comparing HardCopy II and Stratix II Companion Revisions 4–28
Generate a HardCopy II Handoff Report 4–29
Archive HardCopy II Handoff Files 4–29
HardCopy II Advisor 4–30
HardCopy II Floorplan View 4–32
HardCopy Stratix Device Support 4–34
Features 4–35
HARDCOPY_FPGA_PROTOTYPE, HardCopy Stratix and Stratix Devices 4–36
HardCopy Design Flow 4–38
The Design Flow Steps of the One-Step Process 4–39
How to Design HardCopy Stratix Devices 4–40
Tcl Support for HardCopy Migration 4–44
Design Optimization and Performance Estimation 4–45
Design Optimization 4–45
Performance Estimation 4–45
Buffer Insertion 4–48
Placement Constraints 4–48
Location Constraints 4–49
LAB Assignments 4–49
LogicLock Assignments 4–50
Checking Designs for HardCopy Design Guidelines 4–51
Altera-Recommended HDL Coding Guidelines 4–51
Design Assistant 4–51
Reports and Summary 4–52
Generating the HardCopy Design Database 4–53
Static Timing Analysis 4–55
Early Power Estimation 4–55
HardCopy Stratix Early Power Estimation 4–55
HardCopy APEX Early Power Estimation 4–56
Tcl Support for HardCopy Stratix 4–56
Targeting Designs to HardCopy APEX Devices 4–57
Conclusion 4–57
Referenced Documents 4–58
Document Revision History 4–59

Section II. Design Guidelines

Chapter 5. Design Recommendations for Altera Devices and the Quartus II Design Assistant

Introduction 5–1
Synchronous FPGA Design Practices 5–2
Fundamentals of Synchronous Design 5–2
Hazards of Asynchronous Design 5–3
Design Guidelines 5–4
Combinational Logic Structures 5–4

Clocking Schemes 5–9
Checking Design Violations Using the Design Assistant 5–15
Quartus II Design Flow with the Design Assistant 5–15
The Design Assistant Settings Page 5–17
Message Severity Levels 5–18
Design Assistant Rules 5–18
Enabling and Disabling Design Assistant Rules 5–37
Viewing Design Assistant Results 5–40
Targeting Clock and Register-Control Architectural Features 5–44
Clock Network Resources 5–44
Reset Resources 5–45
Register Control Signals 5–46
Conclusion 5–46
Referenced Documents 5–46
Document Revision History 5–47

Chapter 6. Recommended HDL Coding Styles

Introduction 6–1
Quartus II Language Templates 6–2
Using Altera Megafunctions 6–3
Instantiating Altera Megafunctions in HDL Code 6–4
Instantiating Megafunctions Using the MegaWizard Plug-In Manager 6–4
Creating a Netlist File for Other Synthesis Tools 6–6
Instantiating Megafunctions Using the Port and Parameter Definition 6–7
Inferring Multiplier and DSP Functions from HDL Code 6–7
Multipliers—Inferring the lpm_mult Megafunction from HDL Code 6–7
Multiply-Accumulators and Multiply-Adders—Inferring altmult_accum and altmult_add Megafunctions from HDL Code 6–10
Inferring Memory Functions from HDL Code 6–13
RAM Functions—Inferring altsyncram and altdpram Megafunctions from HDL Code 6–14
ROM Functions—Inferring altsyncram and lpm_rom Megafunctions from HDL Code 6–31
Shift Registers—Inferring the altshift_taps Megafunction from HDL Code 6–33
Coding Guidelines for Registers and Latches 6–37
Register Power-Up Values in Altera Devices 6–37
Secondary Register Control Signals Such as Clear and Clock Enable 6–39
Latches 6–43
General Coding Guidelines 6–48
Tri-State Signals 6–49
Adder Trees 6–50
State Machines 6–52
Multiplexers 6–60
Cyclic Redundancy Check Functions 6–69
Comparators 6–71
Counters 6–73
Designing with Low-Level Primitives 6–73
Conclusion 6–74
Referenced Documents 6–74
Document Revision History 6–75

Section III. Synthesis

Chapter 7. Synplicity Synplify and Synplify Pro Support

Introduction 7–1
Altera Device Family Support 7–2
Design Flow 7–3
Output Netlist File Name and Result Format 7–7
Synplify Optimization Strategies 7–8
Implementations in Synplify Pro 7–8
Timing-Driven Synthesis Settings 7–9
FSM Compiler 7–11
Optimization Attributes and Options 7–12
Altera-Specific Attributes 7–15
Exporting Designs to the Quartus II Software Using NativeLink Integration 7–17
Running the Quartus II Software from within the Synplify Software 7–18
Using the Quartus II Software to Run the Synplify Software 7–19
Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script 7–19
Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File 7–20
Passing Constraints to the Quartus II Software using Tcl Commands 7–22
Guidelines for Altera Megafunctions and Architecture-Specific Features 7–32
Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager 7–33
Inferring Altera Megafunctions from HDL Code 7–37
Incremental Compilation and Block-Based Design 7–44
Hierarchy and Design Considerations with Multiple VQM Files 7–46
Creating a Design with Separate Netlist Files 7–46
Creating a Design with Multiple VQM Files Using Synplify Pro MultiPoint Synthesis 7–47
Generating a Design with Multiple VQM Files Using Black Boxes 7–54
Conclusion 7–60
Referenced Documents 7–61
Document Revision History 7–61

Chapter 8. Quartus II Integrated Synthesis

Introduction 8–1
Design Flow 8–2
Language Support 8–5
Verilog HDL Support 8–5
VHDL Support 8–10
AHDL Support 8–12
Schematic Design Entry Support 8–13
State Machine Editor 8–13
Design Libraries 8–14
Using Parameters/Generics 8–18
Incremental Synthesis and Incremental Compilation 8–23
Partitions for Preserving Hierarchical Boundaries 8–23
Quartus II Synthesis Options 8–24
Setting Synthesis Options 8–25

Optimization Technique 8–30
Speed Optimization Technique for Clock Domains 8–30
PowerPlay Power Optimization 8–31
Restructure Multiplexers 8–32
State Machine Processing 8–34
Manually Specifying State Assignments Using the syn_encoding Attribute 8–35
Manually Specifying Enumerated Types Using the enum_encoding Attribute 8–38
Safe State Machines 8–40
Power-Up Level 8–42
Power-Up Don't Care 8–43
Remove Duplicate Registers 8–44
Remove Redundant Logic Cells 8–44
Preserve Registers 8–44
Disable Register Merging/Don't Merge Register 8–45
Noprune Synthesis Attribute/Preserve Fan-out Free Register Node 8–46
Keep Combinational Node/Implement as Output of Logic Cell 8–47
Don't Retime, Disabling Synthesis Netlist Optimizations 8–48
Don't Replicate, Disabling Synthesis Netlist Optimizations 8–49
Maximum Fan-Out 8–50
Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable 8–51
Megafunction Inference Control 8–52
RAM Style and ROM Style—for Inferred Memory 8–55
Turning Off Add Pass-Through Logic to Inferred RAMs/ no_rw_check Attribute Setting 8–57
RAM Initialization File—for Inferred Memory 8–59
Multiplier Style—for Inferred Multipliers 8–59
Full Case 8–62
Parallel Case 8–63
Translate Off and On / Synthesis Off and On 8–65
Ignore translate_off and synthesis_off Directives 8–65
Read Comments as HDL 8–66
Use I/O Flipflops 8–67
Specifying Pin Locations with chip_pin 8–68
Using altera_attribute to Set Quartus II Logic Options 8–70
Analyzing Synthesis Results 8–73
Analysis and Synthesis Section of the Compilation Report 8–73
Project Navigator 8–74
Analyzing and Controlling Synthesis Messages 8–74
Quartus II Messages 8–74
VHDL and Verilog HDL Messages 8–75
Node-Naming Conventions in Quartus II Integrated Synthesis 8–79
Hierarchical Node-Naming Conventions 8–79
Node-Naming Conventions for Registers (DFF or D Flipflop Atoms) 8–80
Register Changes During Synthesis 8–81
Preserving Register Names 8–84
Node-Naming Conventions for Combinational Logic Cells 8–84

Preserving Combinational Logic Names 8–86
Scripting Support 8–86
Adding an HDL File to a Project and Setting the HDL Version 8–87
Quartus II Synthesis Options 8–88
Assigning a Pin 8–89
Creating Design Partitions for Incremental Compilation 8–90
Conclusion 8–91
Referenced Documents 8–91
Document Revision History 8–92

Chapter 9. Mentor Graphics LeonardoSpectrum Support

Introduction 9–1
Design Flow 9–2
Optimization Strategies 9–5
Timing-Driven Synthesis 9–5
Other Constraints 9–6
Timing Analysis with the Leonardo-Spectrum Software 9–8
Exporting Designs Using NativeLink Integration 9–9
Generating Netlist Files 9–9
Including Design Files for Black-Boxed Modules 9–9
Passing Constraints with Scripts 9–9
Integration with the Quartus II Software 9–10
Guidelines for Altera Megafunctions and LPM Functions 9–10
Inferring Multipliers and DSP Functions 9–12
Controlling DSP Block Inference 9–13
Block-Based Design with the Quartus II Software 9–19
Hierarchy and Design Considerations 9–20
Creating a Design with Multiple EDIF Files 9–21
Generating Multiple EDIF Files Using Black Boxes 9–25
Incremental Synthesis Flow 9–31
Conclusion 9–34
Referenced Documents 9–34
Document Revision History 9–35

Chapter 10. Mentor Graphics Precision RTL Synthesis Support

Introduction 10–1
Device Family Support 10–2
Design Flow 10–2
Creating a Project and Compiling the Design 10–6
Creating a Project 10–6
Compiling the Design 10–7
Mapping the Precision Synthesis Design 10–7
Setting Timing Constraints 10–8
Setting Mapping Constraints 10–9
Assigning Pin Numbers and I/O Settings 10–9
Assigning I/O Registers 10–10
Disabling I/O Pad Insertion 10–11

Controlling Fan-Out on Data Nets 10–12
Synthesizing the Design and Evaluating the Results 10–13
Obtaining Accurate Logic Utilization and Timing Analysis Reports 10–13
Exporting Designs to the Quartus II Software Using NativeLink Integration 10–14
Running the Quartus II Software from within the Precision RTL Software 10–14
Running the Quartus II Software Manually Using the Precision RTL Synthesis-Generated Tcl Script 10–16
Using Quartus II Software to Launch the Precision RTL Synthesis Software 10–17
Passing Constraints to the Quartus II Software 10–17
Megafunctions and Architecture-Specific Features 10–23
Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager 10–24
Inferring Altera Megafunctions from HDL Code 10–25
Incremental Compilation and Block-Based Design 10–32
Hierarchy and Design Considerations 10–34
Creating a Design with Separate Netlist Files 10–34
Creating Quartus II Projects for Multiple EDIF Files 10–39
Conclusion 10–41
Referenced Documents 10–42
Document Revision History 10–42

Chapter 11. Synopsys Design Compiler FPGA Support

Introduction 11–1
Design Flow Using the DC FPGA Software and the Quartus II Software 11–2
Setup of the DC FPGA Software Environment for Altera Device Families 11–3
Megafunctions and Architecture-Specific Features 11–5
Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager 11–6
Clear Box Methodology 11–6
Black Box Methodology 11–9
Inferring Altera Megafunctions from HDL Code 11–11
Reading Design Files into the DC FPGA Software 11–13
Selecting a Target Device 11–15
Timing and Synthesis Constraints 11–16
Compilation and Synthesis 11–18
Reporting Design Information 11–20
Saving Synthesis Results 11–21
Exporting Designs to the Quartus II Software 11–22
write_fpga Command 11–22
write and write_par_constraint Commands 11–23
Using Tcl Scripts with Quartus II Software 11–23
Place and Route with the Quartus II Software 11–25
Formality Software Support 11–26
Conclusion 11–26
Referenced Documents 11–26
Document Revision History 11–27

Chapter 12. Analyzing Designs with Quartus II Netlist Viewers

Introduction 12–1

When to Use Viewers: Analyzing Design Problems 12–2
Quartus II Design Flow with Netlist Viewers 12–3
RTL Viewer Overview 12–4
State Machine Viewer Overview 12–6
Technology Map Viewer Overview 12–6
Introduction to the User Interface 12–7
Schematic View 12–8
Hierarchy List 12–16
State Machine Viewer 12–18
Navigating the Schematic View 12–21
Traversing and Viewing the Design Hierarchy 12–21
Viewing Contents of Atom Primitives in the Technology Map Viewer 12–22
Viewing the Properties of Instances and Primitives 12–24
Viewing LUT Representations in the Technology Map Viewer 12–24
Zooming and Magnification 12–26
Partitioning the Schematic into Pages 12–28
Customizing the Schematic Display in the RTL Viewer 12–31
Grouping Combinational Logic into Logic Clouds 12–32
Filtering in the Schematic View 12–34
Filter Sources Command 12–35
Filter Destinations Command 12–35
Filter Sources and Destinations Command 12–36
Filter Between Selected Nodes Command 12–36
Filter Selected Nodes and Nets Command 12–37
Filter Bus Index Command 12–37
Filter Command Processing 12–37
Filtering Across Hierarchies 12–38
Expanding a Filtered Netlist 12–40
Reducing a Filtered Netlist 12–41
Probing to Source Design File and Other Quartus II Windows 12–42
Moving Selected Nodes to Other Quartus II Windows 12–42
Probing to the Viewers from Other Quartus II Windows 12–44
Viewing a Timing Path 12–45
Other Features in the Schematic Viewer 12–47
Tooltips 12–47
Radial Menu 12–50
Rollover 12–52
Displaying Net Names 12–53
Displaying Node Names 12–53
Find Command 12–53
Exporting and Copying a Schematic Image 12–55
Printing 12–55
Debugging HDL Code with the State Machine Viewer 12–56
Simulation of State Machine Gives Unexpected Results 12–56
Conclusion 12–60
Document Revision History 12–60

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook, Volume 1*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1. Design Planning with the Quartus II Software
Revised: *October 2007*
Part number: *QII51016-7.2.0*

Chapter 2. Quartus II Incremental Compilation for Hierarchical and Team-Based Design
Revised: *October 2007*
Part number: *QII51015-7.2.0*

Chapter 3. Quartus II Design Flow for MAX+PLUS II Users
Revised: *October 2007*
Part number: *QII51002-7.2.0*

Chapter 4. Quartus II Support for HardCopy Series Devices
Revised: *October 2007*
Part number: *QII51004-7.2.0*

Chapter 5. Design Recommendations for Altera Devices and the Quartus II Design Assistant
Revised: *October 2007*
Part number: *QII51006-7.2.0*

Chapter 6. Recommended HDL Coding Styles
Revised: *October 2007*
Part number: *QII51007-7.2.0*

Chapter 7. Synplicity Synplify and Synplify Pro Support
Revised: *October 2007*
Part number: *QII51009-7.2.0*

Chapter 8. Quartus II Integrated Synthesis
Revised: *October 2007*
Part number: *QII51008-7.2.0*

Chapter 9. Mentor Graphics LeonardoSpectrum Support
Revised: *October 2007*
Part number: *QII51010-7.2.0*

Chapter 10. Mentor Graphics Precision RTL Synthesis Support
Revised: *October 2007*
Part number: *QII51011-7.2.0*

Chapter 11. Synopsys Design Compiler FPGA Support
Revised: *October 2007*
Part number: *QII51014-7.2.0*

Chapter 12. Analyzing Designs with Quartus II Netlist Viewers
Revised: *October 2007*
Part number: *QII51013-7.2.0*

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 7.2.

How to Contact Altera

For the most up-to-date information about Altera products, refer to the following table.

Information Type	Contact *(1)*
Technical support	www.altera.com/mysupport/
Technical training	www.altera.com/training/ custrain@altera.com
Product literature	www.altera.com/literature/
Altera literature services	literature@altera.com *(1)*
FTP site	ftp.altera.com

Note to table:
(1) You can also contact your local Altera sales office or sales representative.

Third-Party Software Product Information

Third-party software products described in this handbook are not Altera products, are licensed by Altera from third parties, and are subject to change without notice. Updates to these third-party software products may not be concurrent with Quartus II software releases. Altera has assumed responsibility for the selection of such third-party software products and its use in the Quartus II 7.2 software release. To the extent that the software products described in this handbook are derived from third-party software, no third party warrants the software, assumes any liability regarding use of the software, or undertakes to furnish you any support or information relating to the software. EXCEPT AS EXPRESSLY SET FORTH IN THE APPLICABLE ALTERA PROGRAM LICENSE SUBSCRIPTION AGREEMENT UNDER WHICH THIS SOFTWARE WAS PROVDED TO YOU, ALTERA AND THIRD-PARTY LICENSORS DISCLAIM ALL WARRANTIES WITH RESPECT TO THE USE OF SUCH THIRD-PARTY SOFTWARE CODE OR DOCUMENTATION IN THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT. For more information, including the latest available version of specific third-party software products, refer to the documentation for the software in question.

Typographic Conventions

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input.` Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
✓, —, N/A	Used in table cells to indicate the following: ✓ indicates a "Yes" or "Applicable" statement; — indicates a "No" or "Not Supported" statement; N/A indicates that the table cell entry is not applicable to the item of interest.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.

Section I. Design Flows

The Altera® Quartus® II, version 7.2 design software provides a complete multi-platform design environment that easily adapts to your specific design needs. The Quartus II software also allows you to use the Quartus II graphical user interface, EDA tool interface, or command-line interface for each phase of the design flow. This section explains the Quartus II, version 7.2 software options that are available for each of these flows.

This section includes the following chapters:

- Chapter 1, Design Planning with the Quartus II Software
- Chapter 2, Quartus II Incremental Compilation for Hierarchical and Team-Based Design
- Chapter 3, Quartus II Design Flow for MAX+PLUS II Users
- Chapter 4, Quartus II Support for HardCopy Series Devices



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

1. Design Planning with the Quartus II Software

QII51016-7.2.0

Introduction

Due to the significant increase in FPGA device densities over the last few years, designs are increasingly complex and may involve multiple designers. The inherent flexibility of advanced FPGAs means that the pin layout, power consumption, and timing performance for each design block are all dependent on the final design implementation. The system architect must resolve these design issues when integrating design blocks, often leading to problems that affect the overall time to market and thereby increase cost. Many potential problems can be solved earlier in the design cycle by selecting the optimal device and programming method, properly planning I/O pin locations, estimating power consumption, selecting appropriate third-party tools, planning for in-system debugging options, performing good design partitioning for incremental compilation, and obtaining early timing estimates.

This chapter discusses these important FPGA design planning issues, provides recommendations, and describes various tools available for Altera® FPGAs to help you improve design productivity. This chapter contains the following sections:

- "Device and Programming/ Configuration Method Selection" on page 1–2
- "Early Planning Tools for Power and I/O" —"Early Power Estimation" on page 1–5
- "Early Pin Planning and I/O Analysis" on page 1–6
- "Selecting Third-Party EDA Tool Flows" on page 1–9
- "Planning for On-Chip Debugging Options" on page 1–11
- "Planning for an Incremental Compilation Flow" on page 1–13
- "Early Timing Estimation" on page 1–19

Before reading the design planning guidelines discussed in this chapter, consider your design priorities: What are the important factors for your design? More device features, density, or performance can increase system cost. Signal integrity and board issues may impact I/O pin locations. Power, timing performance, and area utilization affect each other, and compilation time is affected by optimizations for these factors. The Quartus® II software optimizes designs for the best average results, but you can change settings to focus on one aspect of the design results and trade off other aspects. Certain tools or debugging options can lead to restrictions in your design flow. If you know what is important in a particular design, this knowledge will help you choose the tools, features, and methodologies that you should use with the design. This chapter

cannot cover every possible consideration for planning a complex FPGA design, but once you understand your design priorities, you can use the design planning issues described here as a guide to help ensure a productive and successful FPGA design flow.



This chapter provides an introduction to various design and planning features in the Quartus® II software. For a general overview of the Quartus II design flow and features, refer to the *Introduction to Quartus II Manual*. For more details about specific Quartus II features and methodologies, this chapter provides references to other appropriate chapters in the *Quartus II Handbook*.

Device and Programming/ Configuration Method Selection

The first stage in design planning is choosing the best device for your application and determining how you want to program or configure the device in your system. These factors affect the rest of your design cycle, including board specification and layout. Most of this planning is performed outside of the Quartus II software, but this section provides a few suggestions to aid in the planning process.

Device Selection

It is important to choose the device family that best suits your design needs. Different families offer different trade-offs, including cost, performance, logic and memory density, I/O density, power utilization, and packaging. You should also consider feature requirements such as I/O standards support, high-speed transceivers, and the number of phase-locked loops (PLLs) available in the device. You can review important features of each device family in the Selector Guides available on the Altera website (**www.altera.com/literature/lit-sg.jsp**). Each device family also has a device handbook or set of data sheets that documents the device features in detail.

Determining the required device density can be a challenging part of the design planning process. Devices with more logic resources and higher I/O counts can implement larger and potentially more complex designs, but may have a higher cost. Select a device that meets your design needs with some safety margin, in case you want to add more logic later in the design cycle or reserve logic and memory for on-chip debugging (refer to "Planning for On-Chip Debugging Options" on page 1–11). Consider needs for specific types of dedicated logic blocks, such as memory blocks of different sizes, or digital signal processing (DSP) blocks to implement certain arithmetic functions.

If you have prior designs targeting Altera devices, you can use their resource utilization as an estimate for your new design. You can compile existing designs in the Quartus II software with the device selection set to

Auto to review the resource utilization and find out which device density fits the design. Note that coding style, device architecture, and the optimization options used in the Quartus II software can significantly affect a design's resource utilization.

To obtain resource utilization estimates for certain configurations of Altera's intellectual property (IP) designs, refer to the User Guides for Altera Megafunctions and IP MegaCores on the IP Megafunctions page on the Altera website (**www.altera.com/literature/lit-ip.jsp**). You can use these numbers to help estimate the resource utilization of your design.

Device Migration Planning

Determine if you want the option of migrating your design to another device density to allow flexibility when the design nears completion, or if you want to migrate to a HardCopy® structured ASIC device when the design reaches volume production. In some cases, designers may target a smaller (and less expensive) device and then move to a larger device if necessary to fit their design. Other designers may prototype their design in a larger device to reduce optimization time and achieve timing closure more quickly, and then migrate to a final smaller device after prototyping. Similarly, many designers compile and optimize their design for an FPGA device before moving to a HardCopy structured ASIC when the design is complete and ready for higher-volume production. If you would like this flexibility, you should specify these migration options in the Quartus II software at the beginning of your design cycle. Specify the target migration devices in the **Migration compatibility** section of the **Device** page in the **Settings** dialog box.

Selecting a migration device has an impact on pin placement because some pins may serve different functions in different device densities or package sizes. When making pin assignments in the Quartus II software, the Pin Migration View in the Pin Planner highlights pins that change function between your migration devices. (Refer to "Early Pin Planning and I/O Analysis" on page 1–6 for more details.) Selecting a migration device may force you to restrict logic utilization to ensure that your design is compatible with a selected HardCopy device. Adding migration devices later in the design cycle is possible, but requires extra effort to check pin assignments, and may require design changes to fit into the new target device. It is much easier to consider these issues early in the design cycle than at the end, when the design is near completion and ready for migration.

In addition, if you are planning to use a HardCopy device, review HardCopy guidelines early in the design cycle for any Quartus II settings that should be used or other restrictions you should consider. It is especially important to use complete timing constraints if you want to migrate to a HardCopy device because of the rigorous verification requirements for structured ASICs.



For more information about timing requirements and analysis for HardCopy designs, refer to the *HardCopy Handbook*.

Programming/Configuration Method Selection

Choosing your programming or configuration method up-front allows system and board designers to determine what companion devices, if any, are needed for your system. Your board layout also depends on the type of programming or configuration method you plan to use for programmable devices. Many programming options use a JTAG interface to connect to the devices, so your design may require a JTAG chain be set up on the board.

The device family handbooks describe the configuration options available for a given device family. For more details about configuration options, refer to the *Configuration Handbook*. For information about programming CPLD devices, refer to your device data sheet or handbook. Programming and configuration of Altera devices includes the following options:

- Using enhanced configuration devices—These devices combine industry-standard flash memory with a feature-rich configuration controller, including device features such as concurrent and dynamic configuration, data compression, clock division, and an external flash memory interface. You can also implement remote and local system updates with enhanced configuration devices.
- Using Flash memory devices with a memory controller, such as an Altera MAX® device—The flash memory controller can interface with a PC or microprocessor to receive configuration data via a parallel port.
- Using the Quartus II Serial Flash Loader (SFL)—This scheme allows you to configure the FPGA and program serial configuration devices using the same JTAG interface.
- Using the Quartus II Parallel Flash Loader (PFL)—This solution quickly retrieves data from a JTAG interface and generates data formatted for the receiving target flash device, significantly reducing the flash device programming time. If your system already contains a common flash interface (CFI) flash memory, you can utilize it for the FPGA configuration storage as well, because the PFL feature supports many common industry-standard flash devices. If you

choose this method, you should check the list of supported flash devices early in your system design cycle and plan accordingly. Refer to *AN 386: Using the MAX II Parallel Flash Loader with the Quartus II Software* for the list of supported Flash devices.

Early Planning Tools for Power and I/O

You can use the Quartus II early power and I/O planning tools to provide information to PCB board and system designers. Providing FPGA device information early in the design process enables earlier planning for power and board design requirements. You can perform early power estimation, as well as early pin planning and analysis, before you have created any source code, or when you have a preliminary version of the design, and then perform the most accurate analysis when the design is complete.

Early Power Estimation

Device power consumption must be accurately estimated to develop an appropriate power budget and to design the power supplies, voltage regulators, heat sink, and cooling system. Power estimation and analysis has two significant planning requirements:

- Thermal planning—You must ensure that the cooling solution is sufficient to dissipate the heat generated by the device. In particular, the computed junction temperature must fall within normal device specifications.
- Power supply planning—Power supplies must provide adequate current to support device operation.

Power consumption in FPGA devices is dependent on the design, providing a challenge during early board specification and layout. The Altera PowerPlay Early Power Estimator spreadsheet allows you to estimate power utilization before the design is complete, by processing information about the device resources that will be used in the design, as well as the operating frequency, toggle rates, and environmental considerations.

If you have an existing design or a partially-completed design, the power estimator file generated by the Quartus II software can provide input to the spreadsheet for your current design (refer to "Early Power Estimator File" on page 1–6).

When the design is complete, the PowerPlay Power Analyzer tool in the Quartus II software provides an accurate estimation of power to help ensure that thermal and supply budgets are not violated.

The PowerPlay Early Power Estimator spreadsheets for each supported device family are available on the Altera website: **(www.altera.com/support/devices/estimator/pow-powerplay.jsp)**.

Estimating power consumption early in the design cycle allows planning of power budgets and avoids surprises for designers developing the PCB.



For more information about power estimation and analysis, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Early Power Estimator File

When entering data into the Early Power Estimator spreadsheet, you must include the device resources, operating frequency, toggle rates, and other parameters. Specifying these values requires familiarity with the design. If you do not have an existing design, estimate the number of device resources used in your design and enter it manually. If you have an existing design or a partially completed design, you can generate a power estimator file.

First, compile your design in the Quartus II software. After compilation is complete, on the Project menu, click **Generate PowerPlay Early Power Estimator File**. This command instructs the Quartus II software to write out a power estimator Comma-Separated Value (**.csv**) file (or a text **[.txt]** file for older device families).

The PowerPlay Early Power Estimator spreadsheet includes the Import Data macro, which parses the information in the power estimation file and transfers it into the spreadsheet. If you do not want to use the macro, you can transfer the data into the Early Power Estimator spreadsheet manually.

If the existing Quartus II project represents only a portion of your full design, you should enter the additional resources used in the final design manually. You can edit the spreadsheet and add additional device resources after importing the power estimation file information.

Early Pin Planning and I/O Analysis

It is important to plan top-level FPGA I/O pins early, so board designers can start developing the PCB design and layout. The FPGA device's I/O capabilities influence pin locations and other types of assignments. In cases where the board design team specifies an FPGA pin-out, it is crucial that the pin locations be verified in the FPGA place-and-route software as soon as possible to avoid the need for board design changes.

Traditionally, designers and system architects could not check the validity of FPGA pin assignments until the design was complete. You can now create a preliminary pin-out for an Altera FPGA using the Quartus II Pin Planner before the source code is designed, based on standard I/O interfaces (such as memory and bus interfaces) and any other I/O-related assignments defined by system requirements. Refer to "Creating a Top-Level Design File for I/O Analysis" on page 1–8. Quartus II I/O Assignment Analysis checks that the pin locations and assignments are supported in the target FPGA architecture. You can use **I/O Assignment Analysis** to validate I/O-related assignments that you make or modify throughout the design process.

The Pin Planner enables easy I/O pin assignment planning, assignment, and validation. Use the Pin Planner Package view to make pin location and other assignments using a device package view instead of pin numbers. The Pads view displays I/O pads in order around the silicon die to help you follow pad distance and pin placement guidelines. With the Pin Planner, you can identify I/O banks, voltage reference (VREF) groups, and differential pin pairings to help you through the I/O planning process. If migration devices are selected (including HardCopy devices), as described in "Device Migration Planning" on page 1–3, the Pin Migration view highlights pins that change function in the migration device when compared to the currently selected device. Selecting pins in the Device Migration view cross-probes to the rest of the Pin Planner, so you can use device migration information when planning your pin assignments. You can also configure board trace models of selected pins for use in "board-aware" signal integrity reports generated with the **Enable Advanced I/O Timing** option. You have the option to use a Microsoft Excel spreadsheet to start the I/O planning process if you normally use a spreadsheet in your design flow, and you can export a Comma-Separated Value (**.csv**) file containing your I/O assignments for spreadsheet use when all pins are assigned.

When planning is complete, the pin location information can be passed to PCB designers. The Pin Planner is tightly integrated with certain PCB design EDA tools, and can read pin location changes from these tools to check the suggested changes. It is important that pin assignments match between the Quartus II software and your schematic and board layout tools to ensure the design works correctly on the board where it is placed, especially if changes to the pin-out must be made. The system architect can use the Quartus II software to pass pin information to team members designing individual logic blocks, for better timing closure when they compile their design. Once the design is complete, the Quartus II Fitter reports should be used for the final sign-off of pin assignments.

Starting FPGA pin planning early—before the HDL design is complete—improves the confidence in early board layouts, reduces the chance of error, and improves the design's overall time to market.



For more information about I/O assignment and analysis, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*. For more information about passing I/O information between the Quartus II software and third-party EDA tools, refer to the *Mentor Graphics PCB Design Tools Support* and *Cadence PCB Design Tools Support* chapters in the *I/O and PCB Tools* section in volume 2 of the *Quartus II Handbook*.

Creating a Top-Level Design File for I/O Analysis

Early in the design process, before the source code is created, the system architect typically has information about the I/O interfaces and IP cores that to used in the design. You can use this information with the **Create/Import Megafunction** feature in the Pin Planner to specify details about the design I/O interfaces.

The Pin Planner interfaces with the MegaWizard® Plug-In Manager, and allows you to create or import custom megafunctions and IP cores that use I/O interfaces. Configure the way in which they are connected to each other by specifying matching node names for selected ports in the **Set Up Top-Level Design File** dialog box. Make any other I/O-related assignments for these interfaces or other design I/O pins in the Pin Planner.

When you have entered as much information as possible, generate a top-level design netlist file using the **Create Top-Level Design File** command. The Pin Planner creates virtual pin assignments for internal nodes, so internal nodes will not be assigned to device pins during compilation. Use the generated netlist to perform I/O Analysis with the **Start I/O Assignment Analysis** command.

You can use the I/O analysis results to change pin assignments or IP parameters and repeat the checking process until the I/O interface meets your design requirements and passes the pin checks in the Quartus II software. When this initial pin planning is complete, you can create a Quartus II Revision based on the Quartus II-generated netlist. You then have a choice for how to proceed: you can use the generated netlist to develop the top-level file for the actual design, or disregard the generated netlist and use the generated Quartus II Settings File (**.qsf**) with the actual design.

Selecting Third-Party EDA Tool Flows

Your complete FPGA design flow may include third-party EDA tools in addition to the Quartus II software. Determine which tools you want to use with the Quartus II software to ensure that they are supported and set up correctly, and that you are aware of any useful features or undesired limitations.

Synthesis Tools

You can synthesize your design using the Quartus II software's integrated synthesis tool or your preferred third-party synthesis tool. Different synthesis tools may give different results. If you want to select the best-performing tool for your application, you can experiment by synthesizing typical designs for your application and coding style and comparing the results. Be sure to perform placement and routing in the Quartus II software to get accurate timing analysis and logic utilization results. Results from synthesis are estimates before place-and-route and do not include logic that is treated as a black box for synthesis (such as megafunctions or Altera IP cores in some synthesis tools). In addition, these estimates do not take into account logic usage reduction achieved in the Quartus II Fitter through register packing or other Quartus II optimizations, such as Physical Synthesis, that may change both timing and resource utilization results.

Altera recommends that you use the most recent version of third-party synthesis tools, because tool vendors are continuously adding new features, fixing tool issues, and enhancing performance for Altera devices. The *Quartus II Release Notes* lists the version of each synthesis tool that is officially supported by that version of the Quartus II software.

Specify your synthesis tool in the New Project Wizard or the **EDA Tools Settings** page of the **Settings** dialog box to use the correct Library Mapping File for your synthesis netlist.

Synthesis tools may offer the capability to create a Quartus II project and pass constraints such as the EDA tool setting, device selection, and timing requirements that you specified in your synthesis project. You can use this capability to save time when setting up your Quartus II project for placement and routing.

If you want to take advantage of an incremental compilation methodology, you should partition your design for synthesis and generate multiple output netlist files. Refer to "Incremental Compilation with Design Partitions" on page 1–14 for more information.



For more information about synthesis tool flows, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Simulation Tools

You can use the built-in Quartus II Simulator to perform quick and easy functional and timing simulations. Altera also provides the ModelSim-Altera simulator with Quartus II license subscriptions, which allows you to take advantage of advanced testbench capabilities and other features. In addition, the Quartus II software can generate timing netlist files to support other third-party simulation tools.

If you use a third-party simulation tool, ensure that you use the software version that is supported with your Quartus II version. The *Quartus II Release Notes* list the version of each simulation tool that is officially supported with that particular version of the Quartus II software. Also ensure that you use the model libraries provided with your Quartus II software version. Libraries can change between versions, which might cause a mismatch with your simulation netlist.

Specify your simulation tool in the **EDA Tools Settings** page of the **Settings** dialog box to generate the appropriate output simulation netlist.



For more information about simulation tool flows, refer to the appropriate chapter in the *Simulation* section in volume 3 of the *Quartus II Handbook*.

Formal Verification Tools

The Quartus II software supports some formal verification flows. Consider whether your desired formal verification flow impacts the design and compilation stages of your design.

Using a formal verification flow can impact performance results because it requires that certain logic optimizations be turned off, such as register retiming, and forces hierarchy blocks to be preserved, which can restrict optimization. Formal verification treats memory blocks as black boxes. Therefore, it is best to keep memory in a separate hierarchy block so that other logic does not get incorporated into the black box for verification. There are other restrictions that may also limit your design, so consult the documentation for details. If formal verification is important to your design, it is easier to plan for limitations and restrictions in the beginning than to make changes later in the design flow.

Specify your formal verification tool in the **EDA Tools Settings** page of the **Settings** dialog box to generate the appropriate output netlist.



For more information about formal verification flows, refer to the appropriate chapter in the *Formal Verification* section in volume 3 of the *Quartus II Handbook*.

Planning for On-Chip Debugging Options

Altera's in-system debugging tools offer different advantages and trade-offs, so different debugging tools may work better for different systems and different designers. It is beneficial to evaluate on-chip debugging options early in your design process, to ensure that your system board, Quartus II project, and design are all set up to support the appropriate options. Planning can reduce time spent during debugging and eliminate the need to make changes later to accommodate your preferred debugging methodologies.

The Quartus II portfolio of verification tools includes the following in-system debugging features:

- SignalProbe incremental routing—This feature makes design verification more efficient by quickly routing internal signals to I/O pins without affecting the design. Starting with a fully routed design, you can select and route signals for debugging to either previously reserved or currently unused I/O pins.
- SignalTap® II Embedded Logic Analyzer—This logic analyzer helps you debug an FPGA design by probing the state of the internal signals in the design without the use of external equipment or extra I/O pins, while the design is running at full speed in an FPGA device. Defining custom trigger-condition logic provides greater accuracy and improves the ability to isolate problems. The SignalTap II Embedded Logic Analyzer does not require external probes or changes to the design files to capture the state of the internal nodes or I/O pins in the design; all captured signal data is conveniently stored in device memory until you are ready to read and analyze the data.
- Logic Analyzer Interface—This interface enables you to connect and transmit internal FPGA signals to an external logic analyzer for analysis. You can use this feature to connect a large set of internal device signals to a small number of output pins for debugging purposes, and allows you to take advantage of advanced features in your external logic analyzer or mixed signal oscilloscope.
- In-System Memory Content Editor—This feature provides read and write access to in-system FPGA memories and constants through the JTAG interface, making it easy to test changes to memory contents and constant values in the FPGA while the device is functioning in a system.
- In-System Sources and Probes—This feature sets up customized register chains to drive or sample the instrumented nodes in your logic design, providing an easy way to input simple virtual stimuli and an easy way to capture the current value of instrumented nodes. You can force trigger conditions set up using the SignalTap II Logic Analyzer, create simple test vectors to exercise your design without the use of external test equipment, and dynamically control run-time control signals with the JTAG chain.

- Virtual JTAG Megafunction—The sld_virtual_jtag megafunction allows you to build your own system-level debugging infrastructure, including both processor-based debugging solutions and debugging tools in software for system-level debugging. The sld_virtual_jtag megafunction can be instantiated directly in your HDL code to provide one or more transparent communication channels to access parts of your FPGA design using the JTAG interface of the device.



For more information about debugging tools, refer to "Referenced Documents" on page 1–20.

If you intend to use any of these features, you may have to plan for the features when developing your system board, Quartus II project, and design. The following paragraphs describe various factors to consider during your design planning stages.

The SignalTap II Embedded Logic Analyzer, Logic Analyzer Interface, In-System Memory Content Editor, In-System Sources and Probes, and Virtual JTAG Megafunction all require JTAG connections to perform in-system debugging. Plan your system and board with JTAG ports that are available for debugging.

The JTAG debugging features also require a small amount of additional logic resources to implement the JTAG hub logic. If you set up the appropriate feature early in your design cycle, you can include these device resources in your early resource estimations to ensure you do not over-fill the device with logic.

The SignalTap II Embedded Logic Analyzer uses device memory to capture data during system operation. Consider reserving device memory to be used during debugging, to ensure that you have enough memory resources to take advantage of this debugging technique.

To use incremental debugging with the SignalTap II Embedded Logic Analyzer, the **Full incremental compilation** option must be turned on. This option is on by default for projects created in the Quartus II software version 6.1 or later, but is not turned on automatically for existing projects. If incremental compilation is not enabled, you must recompile the entire design when you want to add debugging functions, or when you make certain changes to SignalTap II settings. Using incremental compilation with the SignalTap II Embedded Logic Analyzer greatly reduces the compilation time required for debugging.

SignalProbe and the Logic Analyzer Interface require I/O pins for debugging. Consider reserving I/O pins for debugging so that you do not have to change the design or board to accommodate debugging signals later. Keep in mind that the Logic Analyzer Interface can multiplex

signals with design I/O pins if required. Ensure that your board supports some kind of debugging mode, where debugging signals do not affect system operation.

If you want to use the Virtual JTAG megafunction for custom debugging applications, you must instantiate it and incorporate it as part of the design process.

The In-System Sources and Probes feature also requires that you instantiate a megafunction in your HDL code. In addition, you have the option to instantiate the SignalTap II Embedded Logic Analyzer as a megafunction so that you can connect it up to nodes in your design manually and ensure that the tapped node names are not changed during synthesis. You can add the debugging block as a separate design partition for incremental compilation to minimize recompilation times.

To use the In-System Memory Content Editor for RAM or ROM blocks or the LPM_CONSTANT megafunction, ensure that you turn on the option **Allow In-System Memory Content Editor to capture and update content independently of the system clock** when you create the memory block in the MegaWizard Plug-In Manager.

Planning for an Incremental Compilation Flow

If you want to take advantage of the compilation-time savings and performance preservation of Quartus II incremental compilation, plan for an incremental compilation flow from the beginning of your design cycle. The following subsections describe the flat compilation flow, where the design hierarchy is flattened without design partitions, and then the incremental compilation flows that use design partitions in top-down, bottom-up, or mixed design methodologies. Incremental compilation flows offer several advantages but require more design planning to ensure good quality of results. The last subsections discuss factors to consider when planning an incremental compilation flow: planning design partitions and creating a design floorplan.



For details about using the incremental compilation flows in the Quartus II software, as well as important guidelines for creating design partitions and a design floorplan, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Flat Compilation Flow with No Design Partitions

In this compilation flow in the Quartus II software, the entire design is compiled together in a "flat" netlist. This flow is used if you do not create any design partitions. Your source code can have hierarchy, but the design is flattened during compilation and all of the design source code

is synthesized and fit in the target device whenever the design is recompiled after any change in the design. By processing the entire design, the software performs all available logic and placement optimizations on the entire design to improve area and performance. You can use debugging tools incrementally, such as the SignalTap II Logic Analyzer, but you do not specify any design partitions to preserve design hierarchy during compilation.

The flat compilation flow is easy to use; you do not have to plan any design partitions. However, because the entire design is recompiled whenever there are any changes to the design, compilation times can be relatively long for large devices. In addition, you may find that the results for one part of the design change when you change a different part of your design.



The full incremental compilation option is turned on by default in the Quartus II software (beginning with version 6.1), so the project is ready for you to create design partitions for incremental compilation. If you do not create any lower-level design partitions, the entire design is considered as a single design partition, and the software uses a flat compilation flow.

Incremental Compilation with Design Partitions

In an incremental compilation flow, the system architect splits a large design into smaller partitions which can be designed separately. Team members can work on partitions independently, which can simplify the design process and reduce compilation time.

When hierarchical design partitions are well chosen and placed in the device floorplan, you can speed up your design compilation time while maintaining or even improving the quality of results.

You may want to use incremental compilation later in the design cycle when you are not interested in improving the majority of the design any further, and want to make changes to, or optimize, one specific block. In this case, you may want to preserve the performance of modules that are unmodified and reduce compilation time on subsequent iterations.

Incremental compilation may also be useful for both reducing compilation time and achieving timing closure. For example, you may want to specify which partitions should be preserved in subsequent incremental compilations and then recompile the other partitions with advanced optimizations turned on.

If a part of your design is not yet complete, you can create an empty partition for the incomplete part of the design while compiling the completed partitions. Then save the results for the complete partitions while you work on the new part of the design.

Alternately, different designers or IP providers may be working on different blocks of the design using a team-based methodology, and you may want to combine these blocks in a bottom-up compilation flow.

In an incremental compilation flow, after you partition the design, the software performs logic synthesis and technology mapping for each partition individually. The Analysis and Synthesis stage reads the project assignments to determine the partition boundaries. If any part of the design changes, Analysis and Synthesis processes the changed partitions and keeps the existing netlist for the unchanged partitions.

If you use a third-party synthesis tool, you should create separate VQM or EDIF netlists for each design partition in your synthesis tool. You may have to create separate projects within your synthesis tool so that the tool synthesizes each partition separately and generates separate output netlist files. Refer to your synthesis tool documentation for information about support for Quartus II incremental compilation. The netlists are then considered the "source files" for incremental compilation. After completion of the Quartus II Analysis and Synthesis step, each partition has one post-synthesis netlist.

The Quartus II Partition Merge step creates a complete netlist that consists of post-synthesis netlists, post-fitting netlists, or both, or netlists imported from lower-level projects, depending on the netlist type you specify for each partition. The Fitter then processes the merged netlist, preserving the placement or placement and routing of unchanged partitions, and refitting only those partitions that have changed.

Top-Down Versus Bottom-Up Incremental Flows

The Quartus II incremental compilation feature supports both top-down and bottom-up compilation flows that are suitable for different design methodologies. You can also combine these flows in a mixed compilation flow. The following subsections briefly describe each of these compilation flows so that you can choose the flow that best meets your design needs.

Top-Down Incremental Compilation Flow

With top-down compilation, one designer or project lead compiles the entire design in the software. Different designers or IP providers can design and verify different parts of the design, and the project lead can add design entities to the project as they are completed. You can also

target optimizations on one part of the design while designating the rest of the design as "empty." Regardless of the source for all the design logic, the project lead compiles and optimizes the top-level project as a whole.

Incremental compilation preserves the compilation results and performance of unchanged partitions in your design, greatly reducing design iteration time by focusing new compilations only on changed design partitions. New compilation results are then merged with the previous compilation results from unchanged design partitions. Additionally, you can target optimization techniques, such as physical synthesis, to specific design partitions while leaving other partitions untouched. You can also use this flow with empty partitions if parts of your design are incomplete or missing.

Bottom-Up and Team-Based Incremental Compilation Flow

Bottom-up design flows allow individual designers to complete the optimization of their design in separate projects and then integrate each lower-level project into one top-level project. Bottom-up methodologies include team-based design flows in which design partitions are created by team members in another location or by third-party IP providers.

Incremental compilation provides export and import features to enable bottom-up design methodologies. Designers of lower-level blocks can export the optimized netlist for their design, along with a set of assignments, such as LogicLock™ regions. The system architect then imports each design block as a design partition in a top-level project.

In bottom-up design flows, it is very important that the system architect provide guidance to designers of lower-level blocks to ensure that each partition uses the appropriate device resources. Because the designs are developed independently, each lower-level designer has no information about the overall design or how their partition connects with other partitions. This lack of information can lead to problems during system integration. The top-level project information, including pin locations, physical constraints, and timing requirements, should be communicated to the designers of lower-level partitions before they start their design.

The system architect can plan design partitions at the top level and use Quartus II incremental compilation to communicate information to lower-level designers through automatically-generated scripts. The Quartus II option **Generate bottom-up design partition scripts** automates the process of transferring top-level project information to lower-level modules. The software provides a project manager interface for managing project information in the top-level design.

The scripts can create Quartus II projects for all the lower-level design blocks and pass all the relevant project assignments. Using these scripts makes it easier for designers of lower-level modules to implement the instructions from the project lead, and avoid conflicts between projects when importing and incorporating the projects into the top-level design. Using this methodology helps reduce the need to further optimize the designs after integration and improves overall designer productivity and team collaboration.

Mixed Incremental Compilation Flow

You can combine top-down and bottom-up compilation flows to take advantage of top-down flows for part of your design, while importing parts of the design that are developed independently.

The top-down flow is generally simpler to perform than its bottom-up counterpart. For example, the need to export and import lower-level designs is eliminated. A top-down approach also provides the design software with information about the entire design, so it can perform global placement optimizations when no part of the design is locked down to a specific location.

In a bottom-up design methodology, you must perform very careful resource balancing and time-budgeting, because the software does not have any information about the other partitions in the top-level design when it compiles individual lower-level partitions. Using bottom-up compilation flows where required, in combination with top-down compilation flows to reduce compilation time and preserve results for other parts of the design, can be an effective way to improve your productivity.

Planning Design Partitions

Partitioning a design for an FPGA requires planning to ensure optimal results when the partitions are integrated, and ensure that each partition is placed well relative to other partitions in the device. Following Altera's recommendations for creating design partitions improves the overall quality of results. For example, registering partition I/O boundaries keeps critical timing paths inside one partition that can be optimized independently. When the design partitions are specified, you can use the **Incremental Compilation Advisor** to ensure that partitions meet Altera's recommendations.

Determining a timing budget before designers develop their individual blocks reduces the chance of timing problems during system integration. If you optimize lower-level partitions separately, any unregistered paths that cross between partitions are not optimized as an entire path. To

ensure that the software correctly optimizes the input and output logic in each partition, you may be required to perform some manual timing budgeting. For each unregistered timing path that crosses between partitions, you should make timing assignments on the corresponding I/O path in each partition to constrain both ends of the path to the budgeted timing delay. Assigning a timing budget for each part of the connection ensures that the software optimizes paths appropriately so they meet the top-level design requirements.

It is important to plan and balance resource utilization. When performing incremental compilation, the software synthesizes each partition separately, with no data about the resources used in other partitions. Therefore, device resources can be overused in the individual partitions during synthesis, and the design may not fit in the target device when the partitions are merged.

In a bottom-up design flow in which designers optimize their lower-level designs and export them to a top-level design, the software also places and routes each partition separately. In some cases, partitions can use conflicting resources when combined at the top level. Balancing resource utilization between the design partitions avoids any problems with conflicting resources when all the partitions are integrated.

Creating a Design Floorplan

To take full advantage of incremental compilation, you should create a design floorplan to avoid conflicts between design partitions, and to ensure that each partition is placed well relative to other partitions. Creating location assignments for each partition ensures that no conflicts occur for locations between different partitions. In addition, a design floorplan helps to avoid a situation in which the Fitter is directed to place or replace a portion of the design in an area of the device where most resources have already been claimed. Without floorplan assignments, this situation can lead to increased compilation time and reduced quality of results.

You can use the Quartus II **Timing Closure Floorplan** or **Chip Planner**, depending on your target device, to create a design floorplan using LogicLock region assignments for each design partition. With a basic design framework for the top-level design, these floorplan editors allow you to view connections between regions, estimate physical timing delays on the chip, and move regions around the device floorplan. When you have compiled the full design, you can also view logic placement and locate areas of routing congestion to improve the floorplan assignments.

Good partition and floorplan design helps lower-level designs meet top-level design requirements when integrated with the rest of the design, reducing the time spent integrating and verifying the timing of the top-level design.



For details about creating placement assignments in the design floorplan, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Early Timing Estimation

It is much less costly to find design issues early in the design cycle than to find problems in the final timing closure stages. Once the first version of the design source code is complete, you may want to perform a quick compilation to create a kind of silicon virtual prototype, or SVP, that you can use to perform timing analysis.

Regardless of your compilation flow, when the design source code is complete you can use the **Start Early Timing Estimate** option to perform a quick compilation and timing analysis of your design. The software chooses a device automatically if required, places any LogicLock regions used to create a floorplan, finds a quick initial placement for all the design logic, and provides a useful estimate of the final design performance. If you have entered timing constraints, timing analysis reports on these constraints.



Early Timing Estimation is supported with both the TimeQuest and Classic Timing Analyzers. Use the TimeQuest Timing Analyzer with Synopsys Design Constraint (SDC) format constraints to enable advanced timing analysis capabilities that are not available in the Classic Timing Analyzer.

Designers of individual blocks in bottom-up design flows can use this feature as they develop the design. Any issues the feature highlights in the lower level design blocks can be communicated to the system architect. Resolving these issues may require allocating additional device resources to the individual block or changing its timing budget.

A top-level designer can also use early timing estimation to prototype the entire design. Incomplete partitions can be marked as empty in an incremental compilation flow, while the rest of the design is compiled to get an early timing estimate and detect any problems with design integration.

A system architect can use early timing estimation along with design partition scripts (as described in "Bottom-Up and Team-Based Incremental Compilation Flow" on page 1–16) to pass additional constraints to lower-level designers, and provide more information about

the other partitions in the design. This information can be especially useful to optimize cross-partition paths. Running early timing estimations helps designers find and resolve design problems during the early design stages.

Conclusion

Modern FPGAs support large, complex designs with fast timing performance. By planning several aspects of your design early in the process, you can reduce unnecessary time spent handling issues in later stages of the process. You can use various features of the Quartus II software to quickly plan your design and achieve the best possible results. Choosing the correct device and programming method, planning I/O pin locations, estimating power consumption, selecting appropriate third-party tools, planning for debugging options, performing good design partitioning, and obtaining early timing estimates all improve productivity, which reduces the design cost and improves the final product's time to market.

Referenced Documents

This chapter references the following documents:

- *AN 386: Using the MAX II Parallel Flash Loader with the Quartus II Software*
- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Cadence PCB Design Tools* chapter in volume 2 of the *Quartus II Handbook*
- *Configuration Handbook*
- *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Design Debugging Using In-System Sources and Probes* chapter in volume 3 of the *Quartus II Handbook*
- *Formal Verification* section in volume 3 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*
- *In-System Updating of Memory and Constants* chapter in volume 3 of the *Quartus II Handbook*
- *Introduction to Quartus II Manual*
- *Mentor Graphics PCB Design Tools Support* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*
- *Simulation* section in volume 3 of the *Quartus II Handbook*

- *sld_virtual_jtag Megafunction User Guide*
- Synthesis section in volume 1 of the Quartus II Handbook

Document Revision History

Table 1–1 shows the revision history for this chapter.

Table 1–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 1–20.	Updated for the Quartus II 7.2 software release.
May 2007 v7.1.0	Updated for the Quartus II 7.1 software release, including: • Expanded Introduction, Device Migration Planning, and Early Pin Planning and Analysis sections. • Added new sections: Selecting Third-Party EDA Tool Flows and Planning for Debug Options. • Other minor changes and reorganization. • Added Referenced Documents.	Updated for the Quartus II 7.1 software release and expanded topic coverage.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Initial release.	—

2. Quartus II Incremental Compilation for Hierarchical and Team-Based Design

QII51015-7.2.0

Introduction

For today's high-density, high-performance FPGA designs, the ability to iterate rapidly during the design and debugging stages is critical. The Quartus® II software delivers advanced technology to create designs for high-density FPGAs. Altera® introduced the FPGA industry's first true incremental design and compilation flow, which provides the following benefits:

- Preserves the results and performance for unchanged logic in your design as you make changes elsewhere.
- Reduces design iteration time by up to 70%, so you can perform more design iterations per day and achieve timing closure efficiently.
- Easy to use in the graphical user interface (GUI).
- Includes Tcl scripting, command-line, and makefile support.
- Facilitates modular hierarchical and team-based design flows using top-down or bottom-up methodologies.
- Supports the Arria™ GX devices, and Stratix® and Cyclone® series of devices. Supports some incremental compilation flows for HardCopy® II devices (for details, refer to "HardCopy Compilation Flows" on page 2–82).

Quartus II incremental compilation is an optional compilation flow. "Choosing a Quartus II Compilation Flow" on page 2–3 provides an overview of the Quartus II design flow with and without incremental compilation to help you decide if you should take advantage of this feature for your project. The remainder of the chapter includes the following sections:

- "Quick Start Guide – Summary of Steps for an Incremental Compilation Flow" on page 2–11
- "Design Partitions" on page 2–17
- "Creating Design Partitions" on page 2–19
- "Setting the Netlist Type for Design Partitions" on page 2–22
- "Creating a Design Floorplan With LogicLock Location Assignments" on page 2–29
- "Exporting and Importing Partitions for Bottom-Up Design Flows" on page 2–32
- "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–46
- "Recommended Design Flows and Compilation Application Examples" on page 2–62
- "Incremental Compilation Restrictions" on page 2–76

- "Scripting Support" on page 2–99
- "Conclusion" on page 2–109

To take advantage of incremental compilation, you organize your design into logical partitions and physical regions for synthesis and fitting (or placement and routing). Incremental compilation preserves the compilation results and performance of unchanged partitions in your design, dramatically reducing design iteration time by focusing new compilations only on changed design partitions. New compilation results are then merged with the previous compilation results from unchanged design partitions. Additionally, you can target optimization techniques, such as physical synthesis, to specific design partitions while leaving other partitions untouched.

Incremental compilation supports two design methodologies: top-down, in which one designer manages a single project for the entire design, and bottom-up, in which each design block can be developed independently. Bottom-up methodologies include team-based design flows in which design partitions are created by team members in another location or by third-party intellectual property (IP) providers. For bottom-up flows, you can generate scripts from the top-level design that pass constraints to lower-level design blocks compiled in separate Quartus II projects.

This chapter contains information to satisfy the following goals:

- Provide an overview of the Quartus II compilation flow and help you decide whether to use incremental compilation
- Describe how to use the Quartus II incremental compilation feature with a quick start guide and then more detailed information
- Provide you with the level of understanding required to make good design decisions to achieve timing closure while speeding up design iterations
- Present several recommended design flows for incremental compilation in the form of examples, along with the rationale behind them and the steps required to carry out the tasks:
  - "Design Flow 1—Changing a Source File for One of Multiple Partitions in a Top-Down Compilation Flow" on page 2–62
  - "Design Flow 2—Optimizing the Placement for One of Multiple Partitions in a Top-Down Compilation Flow" on page 2–63
  - "Design Flow 3—Preserving One Critical Partition in a Multiple-Partition Design in a Top-Down Compilation Flow" on page 2–64
  - "Design Flow 4—Placing All but One Critical Partition in a Multiple-Partition Design in a Top-Down Compilation Flow" on page 2–65
  - "Design Flow 5—Implementing a Team-Based Bottom-Up Design Flow" on page 2–67

- "Design Flow 6—Performing Design Iteration in a Bottom-Up Design Flow" on page 2–71
- "Design Flow 7—Creating Hard-Wired Macros for IP Reuse" on page 2–73

Choosing a Quartus II Compilation Flow

Quartus II incremental compilation enhances the standard Quartus II design flow by allowing you to reuse satisfactory results from previous compilations and save compilation time. This section outlines the flat compilation flow with no design partitions and the incremental flow, and explains the differences. The section explains when a flat compilation flow is satisfactory, and highlights some of the reasons you might want to create design partitions and use the incremental flow.

The full incremental compilation option is turned on by default in the Quartus II software, so the project is ready for you to create design partitions for incremental compilation. If you do not create any design partitions, the software uses a flat compilation flow.

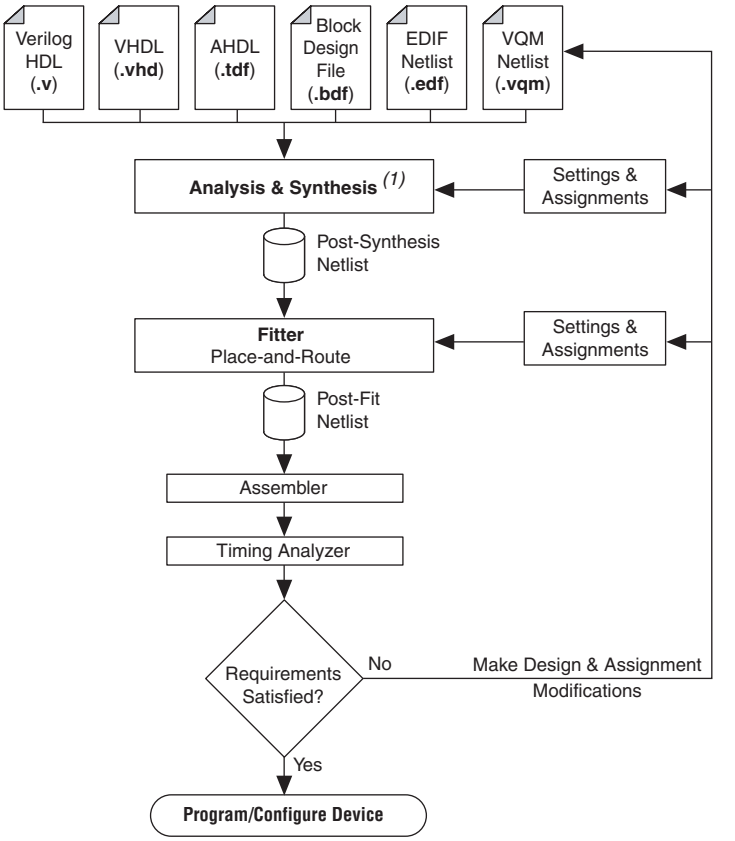
Flat Compilation Flow with No Design Partitions

The standard Quartus II compilation flow consists of the following essential modules:

- **Analysis and Synthesis**—performs logic synthesis to minimize the design logic and performs technology mapping to implement the design logic using device resources such as logic elements. This stage also generates the project database that integrates the design files (including netlists from third-party synthesis tools). When you are using EDIF or VQM netlists created by third-party synthesis tools, the Analysis and Synthesis stage performs logic synthesis and technology mapping only for black boxes and Altera megafunctions.
- **Fitter**—places and routes the logic of a design into a device.
- **Assembler**—converts the Fitter's device, logic, and pin assignments into programming files for the device.
- **Timing Analyzer**—analyzes and validates the timing performance of all the logic in a design.

Figure 2–1 shows a block diagram of the Quartus II design flow with no design partitions.

Figure 2–1. Quartus II Design Flow with No Design Partitions



Note to Figure 2–1:

(1) When you are using EDIF or VQM netlists created by third-party EDA synthesis tools, the Analysis and Synthesis stage of the compilation is performed to create the design database, but logic synthesis and technology mapping are performed only for black boxes and Altera megafunctions.

In any Quartus II compilation flow, you can use smart compilation to allow the compiler to determine which compiler modules are required based on the changes made to the design since the last smart compilation, and then skip any modules that are not required. For example, when smart compilation is selected, the compiler skips the Analysis & Synthesis module if the design source files were unchanged. Smart compilation skips only entire compiler stages. It cannot make incremental changes

within a given stage of the compilation flow. To turn on smart compilation, on the Assignments menu, click **Settings**. In the Category list, select **Compilation Process Settings** and click **Use Smart Compilation**.

In the default flat compilation flow, all of the source code is processed with the Analysis & Synthesis module, and all the logic is placed by the Fitter module whenever the design is recompiled after a change in any part of the design. One reason for this behavior is to obtain optimal quality of results. By processing the entire design, the compiler can perform global optimizations to improve area and performance.

You can use a flat compilation flow for small designs, such as designs in CLPD devices or low-density FPGA devices, when the timing requirements are met easily with a push-button compilation. A flat design is satisfactory when compilation time and preserving results for timing closure are not concerns.

Incremental Compilation Flow with Design Partitions

There are many situations in which an incremental compilation flow is more desirable than the simple flat compilation flow. Using an incremental flow allows you to preserve the results and performance for unchanged logic in your design as you make changes elsewhere. It reduces design iteration time by up to 70%, allowing you to perform more design iterations per day and achieve timing closure more efficiently. Incremental compilation is recommended for large designs and high device densities, as well as designs that require high performance relative to the speed of the device architecture. The feature also facilitates team-based design environments, allowing designers to create and optimize design blocks independently.

In conventional FPGA design, as described in the previous section, a hierarchical design is flattened into a single netlist before logic synthesis and fitting, and the entire design is recompiled every time the design changes. To use the Quartus II incremental compilation flow, you start by splitting your design along any of its hierarchical boundaries into blocks called design partitions. Refer to "Design Partitions" on page 2–17 for more details. The Quartus II software synthesizes each individual hierarchical design partition separately, then merges the partitions into a complete netlist for subsequent stages of the compilation flow. When recompiling the design, you can choose to use source code, post-synthesis results, or post-fitting results for each partition. If you want to preserve the Fitter results, you can choose to keep just the Fitter netlist, keep the placement results, or keep both the placement and routing results.

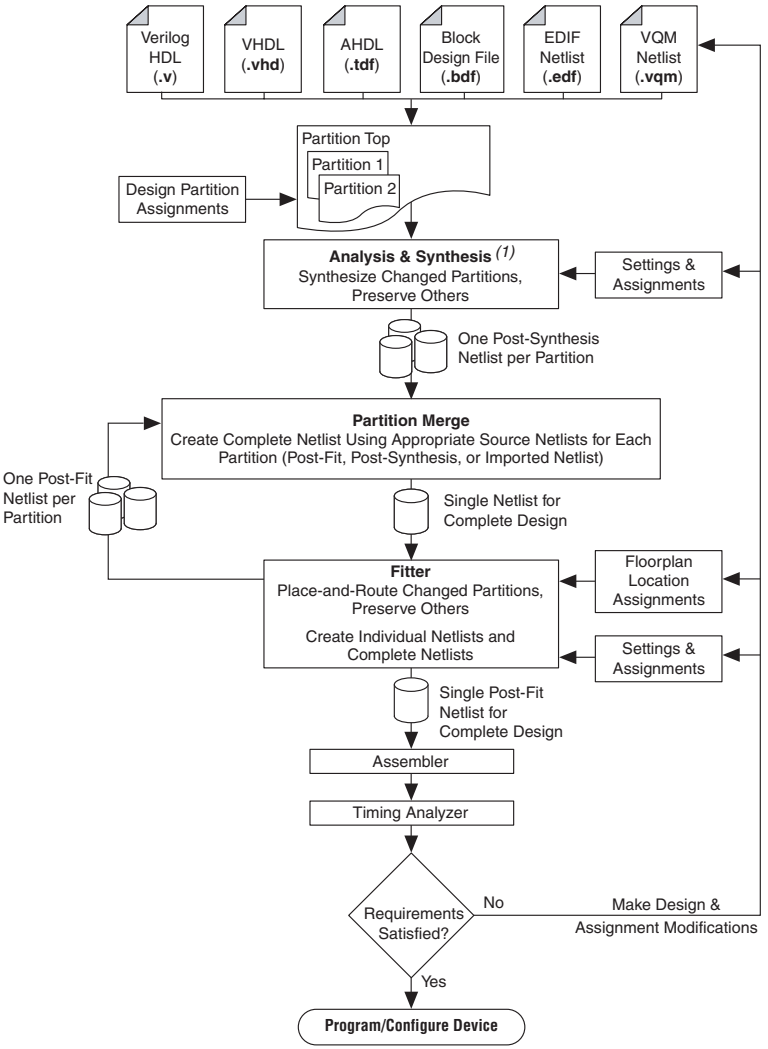
You may want to use incremental compilation later in the design cycle when you are not interested in improving the majority of the design any further, and want to make changes to or optimize one specific block. In this case, you may want to preserve the performance of modules that are unmodified and to reduce compilation time on subsequent iterations. There are also situations in which incremental compilation is useful both for reducing compilation time and for achieving timing closure. For example, you may want to specify which partitions should be preserved in subsequent incremental compilations, and then recompile the other partitions with advanced optimizations turned on.

You might also have part of your design that is not yet complete, for which you can create an empty partition while compiling the completed partitions, and then save the results for the complete partitions while you work on the new part of the design. Alternatively, different designers or IP providers may be working on different blocks of the design using a team-based methodology, and you might want to combine them in a bottom-up compilation flow. In these cases, the Fitter can perform placement and routing on each partition independently.

If you want to use the incremental compilation feature at any point in your design flow, it is beneficial to start planning for incremental compilation from the start of your design development. It is easier to accommodate the guidelines for partitioning and creating a floorplan if you start planning at the beginning of your design cycle. Refer to "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–46 for more information. For more detailed examples that describe recommended design flows to take advantage of the incremental compilation features, refer to "Recommended Design Flows and Compilation Application Examples" on page 2–62.

Figure 2–2 shows a block diagram of the Quartus II design flow using incremental compilation with design partitions.

Figure 2–2. Quartus II Design Flow Using Incremental Compilation



Note to Figure 2–2:

(1) When you are using EDIF or VQM netlists created by third-party EDA synthesis tools, the Analysis and Synthesis stage of the compilation is performed to create the design database, but logic synthesis and technology mapping are performed only for black boxes and Altera megafunctions.

In this flow, Analysis and Synthesis reads the project assignments to determine the partition boundaries, and performs logic synthesis and technology mapping for each partition individually.

The diagram in Figure 2–2 shows a top-level partition and two lower-level partitions. If any part of the design changes, Analysis and Synthesis processes the changed partitions and keeps the existing netlists for the unchanged partitions. After completion of Analysis and Synthesis, there is one post-synthesis netlist for each partition.

The partition merge step creates a single, complete netlist that consists of post-synthesis netlists, post-fit netlists, and netlists imported from lower-level projects, depending on the netlist type you specify for each partition.

The Fitter then processes the merged netlist, preserving the placement or placement and routing of unchanged partitions, refitting only those partitions that have changed. The Fitter generates the complete netlist for use in further stages of the compilation flow, including timing analysis and programming file generation. It also generates individual netlists for each partition so that the partition merge step can use the post-fit netlist to preserve the placement and routing of a partition if you specify to do so in future compilations.

If the design does not meet its requirements (functionality, timing, or area), you can make changes to the design and recompile. The Quartus II software does not resynthesize or refit unchanged partitions that have a netlist type assignment that specifies the use of a post-synthesis or post-fit netlist, respectively.

For more information about using the incremental compilation feature, refer to the "Quick Start Guide – Summary of Steps for an Incremental Compilation Flow" on page 2–11.

See Table 2–1 for a summary of the impact of incremental compilation on your compilation results.

Table 2–1. Summary of the Impact of Full Incremental Compilation (Part 1 of 2)

Characteristic	Impact of Full Incremental Compilation
Compilation Time Savings	Typically saves 50-70% of compilation time when post-fit netlists are preserved; savings in both Quartus II integrated synthesis and the Fitter.
Performance Preservation	Excellent when critical paths are contained within a partition, because you can preserve post-fitting information for unchanged partitions.
Node Name Preservation	Preserves post-fitting node names for unchanged partitions.

Table 2–1. Summary of the Impact of Full Incremental Compilation (Part 2 of 2)

Characteristic	Impact of Full Incremental Compilation
Area Changes	Area might increase because cross-boundary optimizations are no longer possible, and placement and register packing are restricted.
f_{MAX} Changes	f_{MAX} might be reduced because cross-boundary optimizations are no longer possible. If the design is partitioned and the floorplan location assignments are created appropriately, no negative impact on f_{MAX}.
Floorplan Creation	Required for critical partitions to ensure the best quality of results when making design changes. Required in bottom-up flows to avoid placement conflicts.
When Design is Resynthesized	When you set the Netlist Type to use the source file. It is also resynthesized automatically any time you make changes to the source code, unless you specify a Post-Fit (Strict) netlist, or it is an imported partition.
When Design is Refit	When you set the Netlist Type to use the source file, a post-synthesis netlist, or a post-fit netlist with a Fitter preservation level of Netlist Only. It is also refit automatically any time you make changes to the source code, unless you specify a Post-Fit (Strict) netlist, or it is an imported partition.

Top-Down versus Bottom-Up Compilation Flows

The Quartus II incremental compilation feature supports both top-down and bottom-up compilation flows. With top-down compilation, one designer or project lead compiles the entire design in the software. Different designers or IP providers can design and verify different parts of the design, and the project lead can add design entities to the project as they are completed. You can use a top-down flow to optimize one block or IP core in which timing is critical before adding the rest of the design. However, one person (generally the project lead or system architect) compiles the top-level project as a whole. Completed parts of the design can have fitting results and performance fixed as other parts of the design are changing.

Bottom-up design flows allow individual designers or IP providers to complete the optimization of their design in separate projects and then integrate each lower-level project into one top-level project. Incremental compilation provides export and import features to enable this design methodology. Designers of lower-level blocks can export the optimized placed and routed netlist for their design, along with a set of assignments such as LogicLock™ regions. The project lead then imports each design block as a design partition in a top-level project.

The following two benefits are associated with a bottom-up design flow:

- It facilitates team-based development
- It permits the reuse of compilation results from another project, with the ultimate goals of performance preservation and compilation time reduction.

A bottom-up design flow also has the following potential drawbacks that require careful planning:

- It may be difficult to achieve timing closure for the full design because you compile the lower-level blocks independently without any information about each other. This problem may be avoided by careful timing budgeting and special design rules, such as always registering the ports at the module boundaries.
- For the same reason, resource budgeting and allocation may be required to avoid resource conflicts and overuse. Floorplan creation is typically very important in a bottom-up flow.

In a bottom-up design flow, the top-level project lead can do much of the design planning, and then pass constraints on to the designers of lower-level blocks. For more information about the export and import operations, and how to use design partition scripts to help with design planning, refer to "Exporting and Importing Partitions for Bottom-Up Design Flows" on page 2–32.

It is important to understand that with the full incremental compilation flow, users who traditionally relied on a bottom-up approach for the sole reason of performance preservation can now employ a top-down approach to achieve the same goal. This ability is important for two reasons. First, a top-down flow is generally simpler to perform than its bottom-up counterpart. For example, the need to export and import lower-level designs is eliminated. Second, a top-down approach provides the design software with information about the entire design so it can perform global placement and routing optimizations.

You can also mix top-down and bottom-up flows within a single project. If the top-level design includes one or more design blocks that are created by different designers or IP providers, you can import those blocks (using a bottom-up methodology) into a project that also includes partitions for a top-down incremental methodology.

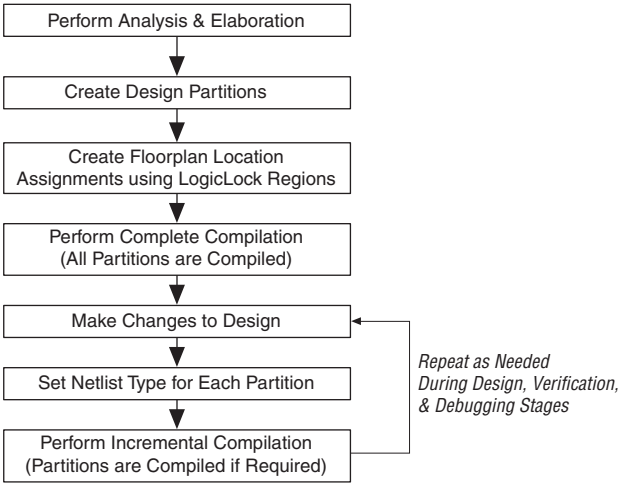
Quick Start Guide – Summary of Steps for an Incremental Compilation Flow

This section provides a summary of the steps required to perform an incremental compilation flow. Detailed descriptions for some of these steps are included in later sections of this chapter. For more examples of design flows that take advantage of the incremental compilation features, refer to "Recommended Design Flows and Compilation Application Examples" on page 2–62.

Top-Down Incremental Compilation Flow

The flow chart in Figure 2–3 illustrates the complete incremental compilation flow using a top-down methodology (all partitions are contained in one top-level project). The following subsections describe the steps in the flow. First, prepare the design for incremental compilation and perform a full compilation. Then proceed to verify or debug your design and make design changes as required. When you perform additional design iterations and recompile your design, you can choose which netlists to reuse and perform incremental compilations.

Figure 2–3. Summary of Top-Down Incremental Compilation Flow



Preparing a Design for Top-Down Incremental Compilation

To set up your design for incremental compilation, use the following general steps:

1. Elaborate the design. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration**, or run any compilation flow that includes this step. Elaboration is part of the synthesis process that identifies your design's hierarchy.

2. Create partitions in your design by applying the **Set as Design Partition** assignment to the appropriate instances.

   Refer to "Design Partitions" on page 2–17 for an explanation of design partitions and what part of your design can be specified as a design partition. Refer to "Creating Design Partitions" on page 2–19 for details about assigning design partitions. For guidelines, refer to "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–46. The most important guidelines include using registers at the I/O boundaries of each partition, and minimizing the number of signals that cross between partitions.

3. Use LogicLock regions to make location assignments for each partition to create a design floorplan. Depending on your design flow and requirements, each partition may be required to be assigned to a physical region on the device. Refer to the section "Creating a Design Floorplan With LogicLock Location Assignments" on page 2–29 for details about these assignments. For guidelines, refer to "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–46.

4. On the Processing menu, click **Start Compilation** to compile the design. The first compilation after making partition and LogicLock assignments is a complete compilation that prepares the design for subsequent incremental compilations.

Compiling a Design Using Incremental Compilation

After compiling the design once and then making changes, you can take advantage of incremental compilation to recompile the changed parts of the design while preserving the results for the unchanged partitions, thus saving time on subsequent compilations. To do this, perform the following general steps:

1. Choose which compilation results you would like to reuse for each partition. To preserve previous placement results for a partition, set the **Netlist Type** assignment for that partition to **Post-Fit**. To preserve routing information as well, set the **Fitter Preservation**

Level to **Placement and Routing**. To save only the synthesis results, set the **Netlist Type** assignment for that partition to **Post-Synthesis**. Partitions with source code changes are recompiled automatically. You can also direct the software to recompile from the source code by choosing the **Source File** netlist type. If you do not want to compile a specific partition at all, set its **Netlist Type** to **Empty**.

For details about setting these partition properties, refer to "Setting the Netlist Type for Design Partitions" on page 2–22.

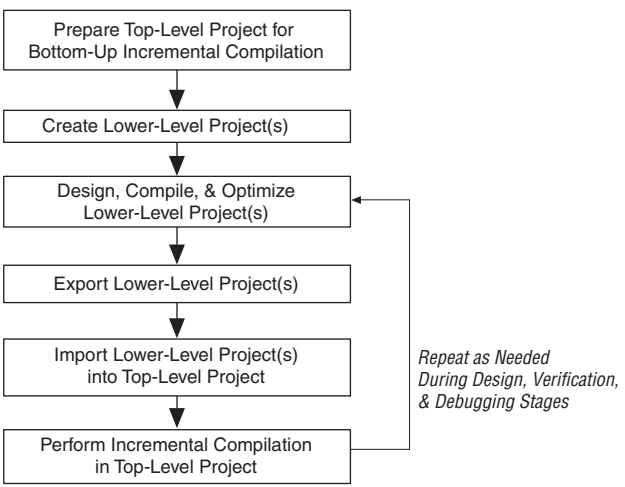
2. Compile the design. When you start a compilation for a partitioned design with incremental compilation turned on, the Quartus II software uses the incremental compilation flow, preserving the results you specified in Step 1.

Bottom-Up Incremental Compilation

The flow chart in Figure 2–4 illustrates the incremental compilation flow using a bottom-up methodology (lower-level partitions are compiled separately before being imported into the top-level project). The following subsections describe the steps involved in the flow.

First, prepare the top-level design for incremental compilation. Then design, optimize, verify, and debug the lower-level projects. Export the lower-level projects, and import them into the top-level design. Finally, compile the entire top-level design.

Figure 2–4. Summary of Bottom-Up Incremental Compilation Flow



Preparing a Design for Bottom-Up Incremental Compilation

The design's project lead or top-level designer should perform the following steps to prepare the design for a successful bottom-up design methodology:

1. Create the top-level Quartus II project that will eventually incorporate the entire design, and apply project-wide settings and global assignments.

    a. Define source code for a "skeleton" of the entire design that defines the hierarchy and the port interfaces for the lower-level designs. The top-level design file must include the top-level entity that instantiates the lower-level blocks you plan to compile in separate Quartus II projects. Include wrapper HDL files for each of these blocks that define at least the port interface. Analysis and Elaboration requires this wrapper file (also known as a "stub" or "black box" file) to connect all the separate design partitions at the top level. For example, in Verilog HDL you should include a module declaration, and in VHDL you should include an entity and architecture declaration. The wrapper file does not have to contain any logic for the design partition.

    b. Create all global assignments, including the device assignment, pin location assignments, and timing assignments, so that the final design meets its requirements. Lower-level project designers can add their own constraints for their partitions as needed, and later provide them to the top-level designer, but the basic constraints can be passed down from the top level to avoid any conflicts and ensure that lower-level projects use the correct assignments.

2. Make design partition assignments for each lower-level design, and set the Netlist Type to Empty for each partition that will be imported. Refer to "Creating Design Partitions" on page 2–19 and "Setting the Netlist Type for Design Partitions" on page 2–22 for details. For guidelines, refer to "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–46.

3. Create LogicLock regions for each of the lower-level partitions to create a design floorplan. Refer to "Creating a Design Floorplan With LogicLock Location Assignments" on page 2–29. For guidelines, refer to "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–46.

4. Optional: Perform a full compilation of the skeleton design and create scripts to pass assignments to lower-level designers. After compilation, on the Project menu, click **Generate Bottom-Up Design Partition Scripts**. Refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–40 for details. Provide each lower-level designer with the generated Tcl file to create their project with the appropriate constraints. If you use makefiles in your design environment, provide the makefile for each partition.

Creating and Compiling Lower-Level Projects

The designer of each lower-level design should create and compile their design in a separate Quartus II project.

If you are creating the project manually, create a new Quartus II project for the subdesign with all the required settings. Create with LogicLock region assignments and global assignments (including clock settings) as specified by the project lead, as well as Virtual Pin assignments for ports which represent connections to core logic instead of external device pins in the top-level module.

If you have a bottom-up design partition script from the top-level designer, source the Tcl script to create the Quartus II project with all the required settings and assignments from the top-level design.

If you are using makefiles, use the **make** command and the makefile provided by the project lead to create a Quartus II project with all the required settings and assignments, and compile the project. Specify the dependencies in the makefile to indicate which source file should be associated with which partition.

Compile and optimize each lower-level design as a separate Quartus II project.

Exporting Lower-Level Projects

When you have achieved the design requirements for the lower-level design, export each design as a partition for the top-level design.

If you are not using makefiles, on the Project menu, use the **Export Design Partition** dialog box to export each lower-level design. Refer to "Exporting a Lower-Level Partition to be Used in a Top-Level Project" on page 2–33. If you want to export only a portion of the design in the lower-level project, refer to "Exporting a Lower-Level Block within a Project" on page 2–35 for instructions. Each lower-level designer must provide the Quartus II Exported Partition file (**.qxp**) to the project lead.

If your design team is using makefiles, the project lead can use the make command with the **master_makefile** to export the lower-level partitions and create Quartus II Exported Partition files, and then import them into the top-level design.

Importing Lower-Level Projects into the Top-Level Project

The project lead then imports the files sent in by the designers of each lower-level subdesign partition.

If you are not using makefiles, on the Project menu, click **Import Design Partition** and specify the partition in the top-level project that is represented by the subdesign Quartus II Exported Partition (QXP) file. Refer to "Importing a Lower-Level Partition Into the Top-Level Project" on page 2–36 for details. Repeat the import process for each partition in the design.

If you are using makefiles, the **master_makefile** command imports each partition into the top-level design. Be sure to specify which source files should be associated with which partition so that the software can rebuild the project if source files change.

For details about which assignments are imported, and how to avoid conflicts, refer to "Importing Assignments and Advanced Import Settings" on page 2–37.

Performing an Incremental Compilation in the Top-Level Project

After you have imported the design partitions that make up the top-level project, you can perform a full compilation. The software compiles imported partitions in the same way as partitions defined in the top-level project. The software recompiles an imported partition only if it has been imported since the last compilation.

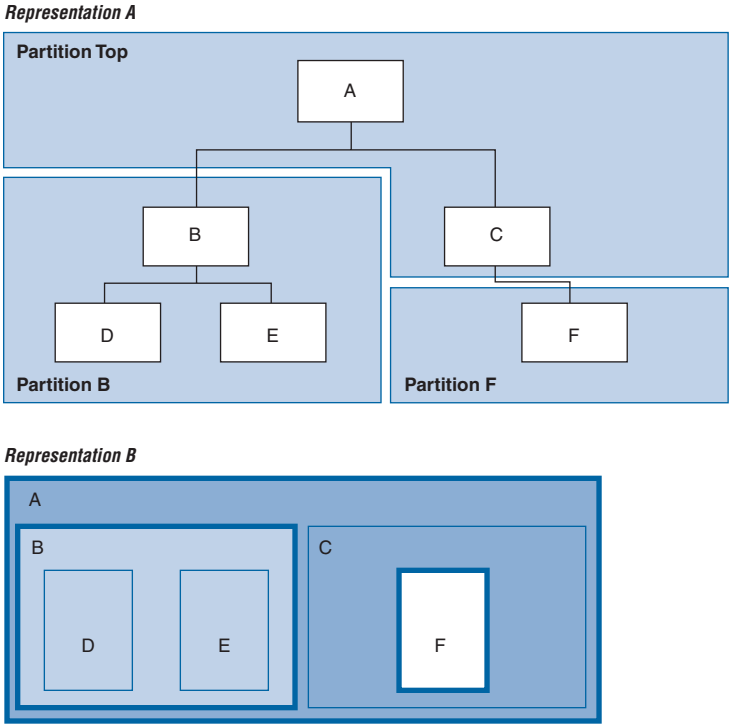
Design Partitions

It is a common design practice to create modular or hierarchical designs in which you develop each design entity separately and then instantiate them in a higher-level entity, forming a complete design. The software does not consider each design entity automatically to be a design partition for incremental compilation; rather, you must designate one or more design hierarchies below the top-level project to be a design partition. Creating partitions prevents the compiler from performing optimizations across partition boundaries, as discussed in "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–46 and illustrated in Figure 2–10. However, this allows for separate synthesis and placement for each partition, making incremental compilation possible.

Partitions must have the same boundaries as hierarchical blocks in the design because a partition cannot be a portion of the logic within a hierarchical entity. When you declare a partition, every hierarchical entity within that partition becomes part of the same partition. You can create new partitions for hierarchical entities within an existing partition, in which case the entities within the new partition are no longer included in the higher-level partition, as described in the following example.

In Figure 2–5, hierarchical entities **B** and **F** form partitions in the complete design, which is made up of entities **A**, **B**, **C**, **D**, **E**, and **F**. The shaded boxes in Representation A indicate design partitions in a "tree" representation of the hierarchy. In Representation B, the lower-level entities are represented inside the higher-level entities, and the partitions are illustrated with different colored shading. The top-level partition, called Top, automatically contains the top-level entity in the design, and contains any logic not defined as part of another partition. The design file for the top level may be just a wrapper for the hierarchical entities below it, or it may contain its own logic. In this example, the partition for top-level entity **A** also includes the logic in one of its lower-level entities, **C**. Because entity **F** is contained in its own partition, it is not treated as part of the top-level partition. Another separate partition, **B**, contains the logic in entities **B**, **D**, and **E**.

Figure 2–5. Partitions in a Hierarchical Design



Design Partition Assignments Compared to Physical Placement Assignments

Design partitions for incremental compilation are logical partitions, different from physical placement assignments in the device floorplan. A logical design partition does not refer to a physical area of the device and does not directly control the placement of instances. A logical design partition sets up a virtual boundary between design hierarchies so each is compiled separately, preventing logical optimizations from occurring between them. The software creates a separate post-synthesis and post-fitting netlist for each partition, which allows the software to reuse the synthesis results or reuse the fitting results (including placement and routing information) in subsequent compilations.

If you preserve the compilation results using a Post-Fit netlist, it is not necessary for you to back-annotate or make any location assignments for specific logic nodes. You should not use the incremental compilation and assignment back-annotation features in the same Quartus II project. The

incremental compilation feature does not use placement "assignments" to preserve placement results; it simply reuses the netlist database that includes the placement information.

You can assign design partitions to physical regions in the device floorplan using LogicLock assignments. Altera recommends using LogicLock regions to improve the quality of results and avoid placement conflicts when performing incremental compilation. LogicLock regions have a size and location on the device floorplan, and you can assign a partition to a physical region to place it in a specific area of the device. Creating floorplan location assignments for design partitions using LogicLock regions is discussed in "Creating a Design Floorplan With LogicLock Location Assignments" on page 2–29.

Creating Design Partitions

To use incremental compilation, you must first split your design into partitions, as described in "Design Partitions" on page 2–17 and "Quick Start Guide – Summary of Steps for an Incremental Compilation Flow" on page 2–11. You can make partition assignments to HDL or schematic design instances, or to VQM or EDIF netlist instances (from third-party synthesis tools). To take advantage of incremental compilation when source files change, the top-level design entity of each partition should have a unique design file. If you define two different entities of separate partitions but they are in the same design file, you cannot maintain incremental compilation because the software would have to recompile both partitions if you changed either entity in the design file.

When you are using a third-party synthesis tool, create a separate netlist file for each partition to allow each partition to be treated incrementally. To create separate netlists for each partition, you may have to create a top-level HDL wrapper file that instantiates the lower-level netlist files and then create separate projects in your synthesis tool for each of the lower-level partitions. In this case, the lower-level blocks should be treated as a black box in the top-level design. Some synthesis tools allow you to create separate netlist files for different design blocks within a single project.



For information about using incremental compilation with third-party synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

For suggestions on determining which parts of your design should be set as design partitions, refer to "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–46.

The full incremental compilation option is turned on by default (for new projects created in the Quartus II software version 6.1 and later), so the project is ready for you to create design partitions.

If full incremental compilation is not turned on when you specify your first partition, a dialog box appears that asks whether you want to enable incremental compilation. Selecting **Full incremental compilation** in this dialog box turns on incremental compilation on the **Incremental Compilation** page under **Compilation Process Settings** in the **Settings** dialog box.

Selecting **Off** on the **Incremental Compilation** page of the **Settings** dialog box does not remove any partition assignments. Partition assignments have no effect on the design if incremental compilation is turned off.

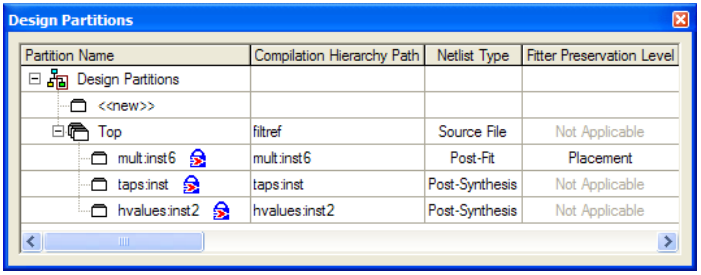
You can create design partitions in the Quartus II GUI with the Design Partitions Window or the Project Navigator.

On the Assignments menu, click **Design Partitions Window** (Figure 2–6) to create your partitions in one of the following ways:

- Create new partitions for one or more instances by dragging and dropping them from the **Hierarchy** tab of the **Project Navigator**, into the Design Partitions window. Using this method, you can create multiple partitions at once.
- Create new partitions by double-clicking the <<*new*>> cell in the Partition Name column. In the **Create New Partitions** dialog box, select the design instance and click **OK**.

To delete partitions in the Design Partitions window, right-click a partition and click **Delete**, or select the partition and press the **Delete** key.

Figure 2–6. Design Partitions Window



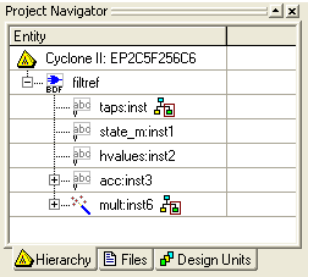
Alternatively, you can use the list of instances under the **Hierarchy** tab in the **Project Navigator** to create and delete design partitions. Right-click on an instance in the **Project Navigator** and click **Set as Design Partition**.



A design partition icon appears next to each instance that is set as a partition (Figure 2–7).

To remove an existing partition assignment, right-click the instance in the **Project Navigator** and click **Set as Design Partition** again. (This process turns off the option.)

Figure 2–7. Project Navigator Showing Design Partitions



Partition Name

When you create a partition, the Quartus II software automatically generates a name based on the instance name and hierarchy path. You can change the name by double-clicking on the partition name in the Design Partitions window, or right-click the partition and click **Rename**. Alternatively, you can right-click the partition in the Design Partitions window and click **Properties** to open the **Design Partition Properties** dialog box. On the **General** tab, enter the new name in the **Name** field.

By renaming your partitions you can avoid referring to them by their hierarchy path, which can sometimes be long. This is especially important when using command-line commands or assignments. Partition names can be from 1 to 1024 characters in length and must be unique. The name can only contain alphanumeric characters and the pipe (|), colon (:), and underscore (_) characters.

Setting the Netlist Type for Design Partitions

The **Netlist Type** property controls the incremental compilation process, as described in "Compiling a Design Using Incremental Compilation" on page 2–12. The **Netlist Type** is a property of each design partition that allows you to specify the type of netlist or source file that the compiler should use as the input for each partition. This property determines which netlist is used by the Partition Merge stage in the next compilation.

To view and modify the **Netlist Type**, on the Assignments menu, click **Design Partitions Window**. Double-click the **Netlist Type** for an entry. Alternatively, right-click on an entry, click **Design Partition Properties**, then modify the **Netlist Type** on the **Compilation** tab.

Table 2–2 describes the different settings for the **Netlist Type** property, explains the behavior of the Quartus II software for each setting, and gives guidance on when to use a certain setting.

Table 2–2. Netlist Type Settings (Part 1 of 3)

Partition Netlist Type	Quartus II Behavior for Partition During Compilation
Source File	Always compiles the partition using the associated design source file(s). You can use this netlist type to recompile a partition from the source code using new synthesis or Fitter settings. If a partition has an associated imported netlist, compiling it with netlist type set to **Source File** removes the imported netlist.
Post-Synthesis	Preserves post-synthesis results for the partition and uses the post-synthesis netlist as long as the following conditions are true: • A post-synthesis netlist is available from a previous synthesis • No change has been made to the associated source files since the previous synthesis Compiles the partition from the source files if there are source changes or if a post-synthesis netlist is not available. Changes to the assignments do not cause recompilation. You can use this netlist type to preserve the synthesis results unless source files change, but refit the partition using any new Fitter settings. If a partition has an associated imported netlist, this setting is not available.

Table 2–2. Netlist Type Settings (Part 2 of 3)

Partition Netlist Type	Quartus II Behavior for Partition During Compilation
Post-Fit	Preserves post-fit results for the partition and uses the post-fit netlist as long as the following conditions are true: ● A post-fit netlist is available from a previous fitting ● No change has been made to the associated source files since the previous fitting Compiles the partition from the source files if there are source changes or if a post-fit netlist is not available. Changes to assignments do not cause recompilation. The **Fitter Preservation Level** specifies what level of information is preserved from the post-fit netlist. For details, refer to "Fitter Preservation Level" on page 2–24. You can use this netlist type to preserve the Fitter results unless source files change. You can also use this netlist type to apply global optimizations, such as Physical Synthesis optimizations, to certain partitions while preserving the fitting results for other partitions. If a partition has an associated imported netlist, this setting is not available.
Post-Fit (Strict)	Always preserves post-fit results for the partition. Uses the post-fit netlist even if changes have been made to the associated source files since the previous fitting. For more information, refer to "Forcing Use of the Post-Fitting Netlist When a Source File has Changed" on page 2–28. The **Fitter Preservation Level** specifies what level of information is preserved from the post-fit netlist. For details, refer to "Fitter Preservation Level" on page 2–24. If a partition has an associated imported netlist, this setting is not available.
Imported	Compiles the design partition using a netlist imported from a Quartus II Exported Partition File **(.qxp)**. The software does not modify or overwrite the original imported netlist during compilation. To preserve changes made to the imported netlist (such as movement of an imported LogicLock region), use the **Post-Fit (Import-based)** setting following a successful compilation with the imported netlist. For additional details, refer to "Exporting and Importing Partitions for Bottom-Up Design Flows" on page 2–32. The **Fitter Preservation Level** specifies what level of information is preserved from the imported netlist. For details, refer to "Fitter Preservation Level" on page 2–24. If you have not imported a netlist for this partition using the **Import Design Partition** command, this setting is not available.

Table 2–2. Netlist Type Settings (Part 3 of 3)

Partition Netlist Type	Quartus II Behavior for Partition During Compilation
Post-Fit (Import-based)	Preserves post-fit results for the partition and uses the post-fit netlist as long as the following conditions are true: ● A post-fit netlist is available from a previous fitting ● No change has been made to the associated imported netlist since the previous fitting Compiles the partition from the imported netlist if the imported netlist changes (which means it has been reimported) or if a post-fit netlist is not available. Changes to assignments do not cause recompilation. The **Fitter Preservation Level** specifies what level of information is preserved from the post-fit netlist. For details, refer to "Fitter Preservation Level". You can use this netlist type to preserve changes to the placement and routing of an imported netlist. If a partition does not have an associated imported netlist, this setting is not available.
Empty	Uses an empty placeholder netlist for the partition and uses virtual pins at the partition boundaries. You can use this netlist type to skip the compilation of a lower-level partition. For more details on the **Empty** setting, refer to "Empty Partitions" on page 2–26.

Fitter Preservation Level

The **Fitter Preservation Level** property specifies which information the compiler will use from a post-fit or imported netlist. The property is only available if the **Netlist Type** is set to **Post-Fit**, **Post-Fit (Strict)**, **Imported**, or **Post-Fit (Import-based)**.

On the Assignments menu, click **Design Partitions Window**. You can view and modify the **Fitter Preservation Level** by double-clicking an entry. You can also right-click and click **Properties**, then edit the **Fitter Preservation Level** on the **Compilation** tab.

Table 2–3 describes the Fitter Preservation Level settings.

Table 2–3. Fitter Preservation Level Settings

Fitter Preservation Level	Quartus II Behavior for Partition During Compilation
Netlist Only	Preserves the netlist atoms of the design partition, but replaces and reroutes the design partition. Unlike a Post-Synthesis netlist, a Post-Fit netlist with the atoms preserved contains any Fitter optimizations, for example, registers duplicated by Physical Synthesis during a previous Fitting. You can use this setting to: ● Preserve Fitter optimizations but allow the software to perform placement and routing again ● Reapply certain Fitter optimizations (that is, physical synthesis) that would otherwise be impossible when the placement is locked down ● Resolve resource conflicts between two imported partitions in a bottom-up design flow
Placement	Preserves the netlist atoms and their placement in the design partition. Re-routes the design partition. This setting saves significant compilation time because the Fitter does not need to re-fit the nodes in the partition. Note that the Fitter may need to modify the placement for timing or legality reasons. This setting might not be available if the netlist type is set to **Imported** and the imported netlist does not contain placement data.
Placement and Routing	Preserves the netlist atoms and their placement and routing in the design partition. The minimum preservation level required to preserve Engineering Change Order (ECO) changes made to the post-fitting netlist and SignalProbe pins added to the design. This setting reduces compilation times compared to Placement only. Note that the Fitter may need to modify the placement and routing for timing or legality reasons. This setting may not be available if the netlist type is set to **Imported** and the imported netlist does not contain routing data.
Placement, Routing, and Tile	Preserves the netlist atoms and their placement and routing in the design partition, as well as the power tile settings of high-speed or low-power. Note that the Fitter may need to modify the placement and routing for timing or legality reasons. This setting is available only for devices with configurable power tiles (currently only Stratix III devices).

Empty Partitions

To set the **Netlist Type** to **Empty**, on the Assignments menu, click **Design Partitions Window**, or double-click an entry, or right-click an entry and click **Design Partition Properties** and select **Empty**. This setting specifies that the Quartus II Compiler should use an empty placeholder netlist for the partition.

You can use the **Empty** setting to skip the compilation of a lower-level partition that is incomplete or missing from the top-level design. You can also use it if you want to compile only some partitions in the design, such as during optimization or if the compilation time is large for one partition and you want to exclude it. This is useful if you want to optimize the placement of a timing-critical block such as an IP core and then lock its placement before adding the rest of your custom logic.

When a partition **Netlist Type** is defined as **Empty**, virtual pins are created at the boundary of the partition. This means that the software temporarily maps I/O pins in the lower-level design entity to internal cells and not to pins during compilation.

Any subpartitions below an empty partition in the design hierarchy are also treated as empty, regardless of their settings.

You can use a design flow in which some partitions are set to **Empty** in a variation of a bottom-up design flow, where you develop pieces of the design separately and then combine them at the top level at a later time. When you implement part of the design without information about the rest of the project, it is impossible for the Compiler to perform global placement optimizations. One way to reduce this effect is to ensure the input and output ports of the partitions are registered whenever possible, as recommended in "Creating Good Design Partitions" on page 2–47.

When you set a design partition to **Empty**, a design file is required in Analysis and Synthesis to specify, at minimum, the port interface information so that it can connect the partition correctly to other logic and partitions in the design. If the design file is missing, you must create a wrapper file (called a black box or hollow-body file) that defines the design block and specifies the input, output, and bidirectional ports. For example, in Verilog HDL you should include a module declaration, and in VHDL you should include an entity and architecture declaration.

What Represents a Source Change for Incremental Compilation?

Any change in any design source file that affects a partition triggers an automatic recompilation of the partition. The only exception is if the partition's **Netlist Type** is set to **Post-Fit (Strict)** – refer to "Forcing Use of the Post-Fitting Netlist When a Source File has Changed" on page 2–28. The Quartus II software uses an internal checksum to determine whether the contents of a source file have changed. Source files are the design files used to create the design, and consist of VHDL files, Verilog HDL files, AHDL files, Block Design Files (**.bdf**), EDIF netlists, VQM netlists, and memory initialization files. Changes in other files such as vector waveform files for simulation do not trigger recompilation.

Changes to certain project-wide assignments, such as changing the device family, also trigger automatic recompilation.

Synthesis and Fitter assignments, including optimization settings, timing assignments, or Fitter location assignments such as pin assignments or LogicLock assignments, do not trigger automatic recompilation in the incremental compilation flow. To recompile a partition with new assignments, change the **Netlist Type** assignment for that partition to one of the following:

- **Source File** to recompile with all new settings
- **Post-Synthesis** to recompile using existing synthesis results but new Fitter settings
- **Post-Fit** with the **Fitter preservation Level** set to **Placement** to rerun routing using existing placement results except for any new routing settings including delay chain settings

The project database folder (**\db**) includes all the netlist information for previous compilations. To avoid unnecessary recompilations, the database files must not be altered or deleted.

If you want to archive or reproduce the project in another location, you can use a Quartus II Archive (**.qar**) file. On the Project menu, click **Archive Project** and turn on **Include database from compilation and simulation** so that compilation results are preserved. To manually create a project archive that preserves compilation results without keeping the entire compilation database, you should keep all source and settings files and create and save a Quartus II Exported Partition (**.qxp**) file for each partition in the design. Refer to "Exporting a Lower-Level Block within a Project" on page 2–35 for more details.

Determining Which Partitions Will Be Recompiled

When design files in a partition have dependencies on other files, changing one file may trigger an automatic recompilation of another file. The **Partition Dependent Files** table in the Analysis and Synthesis report lists the design files that contribute to each design partition. You can use this table to determine which partitions will be recompiled when a specific file is changed.

For example, if a design has files **a.v** that contains entity **a**, **b.v** that contains entity **b**, and **c.v** that contains entity **c**, then the **Partition Dependent Files** table for the partition containing entity **a** lists file **a.v**, the table for the partition containing entity **b** lists file **b.v**, and the table for the partition containing entity **c** lists file **c.v**. Any dependencies are transitive, so if file **a.v** depends on **b.v**, and **b.v** depends on **c.v**, then the entities in file **a.v** depend on files **b.v** and **c.v**. In this case, files **b.v** and **c.v** are listed in the report table as dependent files for the partition containing entity **a**.

If you define module parameters in a higher-level module, you will create file dependencies. The Quartus II software checks the parameter values when determining which partitions require resynthesis. If you change a parameter in a higher-level module that affects a lower-level module, the lower-level module will be resynthesized.

If a design contains common files, such as a file **includes.v** that is referenced in each entity by the command `` `include includes.v ``, then all partitions are dependent on this file. A change to **includes.v** causes the entire design to be recompiled. The VHDL statement `use work.all` also typically results in unnecessary recompilations, because it makes all entities in the work library visible in the current entity, which results in the current entity being dependent on all other entities in the design.

To avoid this type of problem, ensure that files common to all entities, such as a common include file, contain only the set of information that is truly common to all entities. Remove `use work.all` statements in your VHDL file or replace them by including only the specific design units needed for each entity.

Forcing Use of the Post-Fitting Netlist When a Source File has Changed

Forcing the use of the post-fitting netlist when the contents of a source file has changed is recommended only for advanced users who thoroughly understand when a partition must be recompiled. You might want to use this assignment, for example, if you are making source code changes but do not want to recompile the partition until you finish debugging a

different partition. To force the Fitter to use a previously generated post-fit netlist even when there are changes to the source files, you can use the **Post-Fit (Strict)** Netlist Type assignment.

Misuse of the **Post-Fit (Strict)** Netlist Type can result in the generation of a functionally incorrect netlist when source design files change. Use caution in applying this assignment.

Creating a Design Floorplan With LogicLock Location Assignments

After you have partitioned the design, create floorplan location assignments for the design as discussed in this section to improve the quality of results when using the full incremental compilation flow. Creating a design floorplan is not a requirement to use an incremental compilation flow, but it is highly recommended in many cases. Floorplan assignments are required if you want to import partition placement results in a bottom-up flow to avoid placement conflicts at the top level. You should also ensure that you have a LogicLock floorplan assignment for any timing-critical blocks that will be recompiled as you make changes to the design. Logic that is not timing-critical can float throughout the device in a top-down compilation flow, so a floorplan assignment might not be required in this case.

The simplest way to create a floorplan for a partitioned design is to create one LogicLock region per partition (including the top-level partition). Initially, you can leave each region with the default settings of Auto size and Floating location to allow the Quartus II software to determine the optimal size and location for the regions. Then, after compilation, use the Fitter-determined size and origin location as a starting point for your design floorplan. Check the quality of results obtained for your floorplan location assignments and make changes to the regions as needed. Alternately, you can perform synthesis, and then set the regions to the required size based on resource estimates. In this case, use your knowledge of the connections between partitions to place the regions in the floorplan.

For more information about why creating a design floorplan is important in many cases, refer to "The Importance of Floorplan Location Assignments in Incremental Compilation" on page 2–55. For guidelines on creating the floorplan, refer to "Creating Good Floorplan Location Assignments" on page 2–57.

To create a LogicLock region for each design partition, use the following general methodology:

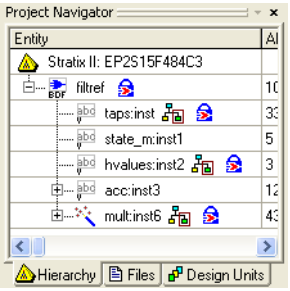
1. On the Assignments menu, click **Design Partitions Window** and ensure that all partitions have their **Netlist Type** set to **Source File** or **Post-Synthesis**. If the **Netlist Type** is set to **Post-Fit**, floorplan location assignments are not used when recompiling the design.

2. Create a LogicLock region for each partition (including the top-level entity, which is automatically considered a partition) using one of the following methods:

   - In the Design Partitions window, right-click on a partition and click **Create New LogicLock Region**. You can highlight multiple (or all) partitions by holding down the **Ctrl** key and clicking on each partition. Then you can choose the option to create a separate LogicLock region for each highlighted partition.
   - Under **Compilation Hierarchy** in the **Project Navigator**, right-click each instance that is denoted as a partition and click **Create New LogicLock Region**.

   A LogicLock icon appears in the Project Navigator next to each instance that is set as a LogicLock region (Figure 2–8).

Figure 2–8. Project Navigator Showing LogicLock Regions



3. On the Processing menu, point to Start and click **Start Early Timing Estimate** to place auto-sized, floating-location LogicLock regions.

   You must perform Analysis and Synthesis and Partition Merge before performing an Early Timing Estimate.

   To run a full compilation instead of the Early Timing Estimate, on the Processing menu, click **Start Compilation**.

4. On the Assignments menu, click **LogicLock Regions Window**, and click on each LogicLock region while holding the **Ctrl** key to select all regions (including the top-level region).

5. Right-click on the last selected LogicLock region, and click **Set Size and Origin to Previous Fitter Results**.



It is important that you use the Fitter-chosen locations only as a starting point to make the regions of a fixed size and location. On average, regions with fixed size and location yield better f_{MAX} than auto-sized regions.

Do not back-annotate the contents of the region, just save the size and origin. Placement is preserved through the use of the post-fit netlist and not any back-annotated content assignments.

6. If required, modify the size and location via the **LogicLock Regions Window** or the **Chip Planner**. For example, make the regions bigger to fill up the device and allow for future logic changes.

7. On the Processing menu, point to **Start** and click **Start Early Timing Estimate** to estimate the timing performance of your design with these LogicLock regions.

8. Repeat steps 6 and 7 until you are satisfied with the quality of results for your design floorplan. On the Processing menu, click **Start Compilation** to run a full compilation.

If you do not want to use auto-sized and floating-location regions, in steps 3–5, you can estimate the size of the regions after synthesis. On the Processing menu, point to **Start**, and choose **Start Analysis & Synthesis**. Right-click on a region in the **LogicLock Regions** dialog box, and choose **Set to Estimated Size**. Then continue with step 6 to modify the size and origin of each region as appropriate.

Taking Advantage of the Early Timing Estimator

The methodology for creating a good floorplan takes advantage of the Early Timing Estimator to enable quick compilations of the design while creating assignments. The Early Timing Estimator feature provides a timing estimate for a design as much as 45 times faster than running a full compilation, yet estimates are, on average, within 11% of final design timing. You can use the Chip Planner to view the "placement estimate" created by this feature, identify critical paths by locating from the timing analyzer reports, and, if necessary, add or modify floorplan constraints. You can then rerun the Early Timing Estimator to quickly assess the impact of any floorplan location assignments or logic changes, enabling rapid iterations on design variants to help you find the best solution.

Exporting and Importing Partitions for Bottom-Up Design Flows

The bottom-up flow refers to the design methodology in which a project is first divided into smaller subdesigns that are implemented as separate projects, potentially by different designers. The compilation results of these lower-level projects are then exported and given to the designer (or the project lead) who is responsible for importing them into the top-level project to obtain a fully functional design.

In a bottom-up design flow, the top-level project lead can do much of the design planning, and then pass constraints on to the designers of lower-level blocks. The bottom-up design partition scripts generated by the Quartus II software can make it easier to plan a bottom-up design, and limit the difficulties that can arise when integrating separate designs. Refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–40 for details.

Refer to "Bottom-Up Incremental Compilation" on page 2–13 in the Quick Start Guide section for an overview of the entire flow. For examples of team-based scenarios, refer to "Bottom-Up Incremental Design Flows" on page 2–67.

This section describes the export and import features provided to support bottom-up compilation flows. The section covers the following topics:

- "Quartus II Exported Partition File (.qxp)"
- "Exporting a Lower-Level Partition to be Used in a Top-Level Project" on page 2–33
- "Exporting a Lower-Level Block within a Project" on page 2–35
- "Importing a Lower-Level Partition Into the Top-Level Project" on page 2–36
- "Importing Assignments and Advanced Import Settings" on page 2–37
- "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–40

Quartus II Exported Partition File (.qxp)

The bottom-up incremental compilation flow uses a file called the Quartus II Exported Partition file (or QXP) to represent lower-level design partitions. The QXP is a binary file that contains compilation results describing the exported design partition and includes a post-fit or post-synthesis netlist, LogicLock regions, and a set of assignments. Note that the QXP file does not contain the original source design files from the lower-level design.

The following sections describe how to generate a QXP file for a lower-level design partition, and how to import the QXP into the top-level project.

Exporting a Lower-Level Partition to be Used in a Top-Level Project

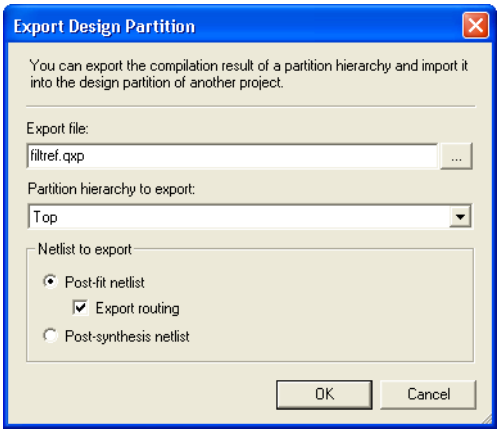
Each lower-level subdesign is compiled as a separate Quartus II project. In each project, use the following guidelines to improve the exporting and importing process:

- If you have a bottom-up design partition script from the top level, source the Tcl script to create the project and all the assignments from the top-level design. Doing so may create many of the assignments described below. Ensure that the LogicLock region uses only the resources allocated by the top-level project lead.
- Ensure that you know which clocks should be allocated to global routing resources so that there are no resource conflicts in the top-level design.
  - Set the **Global Signal** assignment to **On** for the high fan-out signals that should be routed on global routing lines.
  - To avoid other signals being placed on global routing lines, on the Assignments menu, click **Settings** and turn off **Auto Global Clock and Auto Global Register Controls** under **More Settings** on the Fitter page of the **Settings** dialog box.
  - Alternatively, you can set the **Global Signal** assignment to **Off** for signals that should not be placed on global routing lines. Placement for LABs depends on whether the inputs to the logic cells within the LAB use a global clock. You may encounter problems if signals do not use global lines in the lower-level design but use global routing in the top level.
- Use the **Virtual Pin** assignment to indicate pins of a subdesign that do not drive pins in the top-level design. This is critical when a subdesign has more output ports than the number of pins available in the target device. Using virtual pins also helps optimize cross-partition paths for a complete design by enabling you to provide more information about the subdesign ports, such as location and timing assignments.
- Because subdesigns are compiled independently without any information about each other, you should provide more information about the timing paths that may be affected by other partitions in the top-level design. You can apply location assignments for each pin to indicate where the port connection will be located after it is incorporated in the top-level design. You can also apply timing assignments to the I/O ports of the subdesign to perform timing budgeting as described in "Timing Budgeting" on page 2–53.

When your subdesign partition has been compiled using these guidelines, and is ready to be incorporated into the top-level design, export a subdesign as a partition using the following steps:

1. In the subdesign project, on the Project menu, click **Export Design Partition**. The **Export Design Partition** dialog box appears (Figure 2–9).

Figure 2–9. Export Design Partition Dialog Box



2. In the **Export file** box, type the name of the Quartus II Exported Partition file (**.qxp**). By default, the directory path and file name are the same as the current project.

3. You can also select the **Partition hierarchy to export**. By default, the **Top** partition (the entire project) is exported, but you can choose to export the compilation result of any partition hierarchy in the project, as described in "Exporting a Lower-Level Block within a Project" on page 2–35. Choose the partition hierarchy from the drop-down box.

4. Under **Netlist to export**, select either **Post-fit netlist** or **Post-synthesis netlist**. The default is **Post-fit netlist**. For post-fit netlists, turn on or off the **Export routing** option as required.

5. Click **OK**. The Quartus II software creates the Quartus II Exported Partition file in the specified directory.

Alternatively, you can set up your project so that the export process is performed every time you compile the design:

1. On the Assignments menu, click **Settings**.
2. In the **Settings** dialog box, under **Compilation Process Settings**, select the **Incremental Compilation** page.
3. Turn on **Automatically export design partition after compilation**.
4. If you want to view or change the default export settings, click the **Export Design Partition Settings** button to open the **Export Design Partition Settings** dialog box (Figure 2–9).
5. In the **Export Design Partition Settings** dialog box, change the settings, if required, as in steps 2-4 in the preceding export procedure. Click **OK**.
6. Click **OK** to close the **Settings** dialog box. During the next full compilation, the software will create the Quartus II Exported Partition file in the specified directory.

Exporting a Lower-Level Block within a Project

Step 3 in "Exporting a Lower-Level Partition to be Used in a Top-Level Project" enables you to create a Quartus II Exported Partition file for a lower-level block within a Quartus II project. When you do this, the command exports the entire hierarchy under the specified partition into the QXP file.

You can use this feature to add test logic around a lower-level block that will be exported as a design partition for a top-level design. You can also instantiate additional design components in a lower-level project so it matches the top-level design environment. For example, you can include a top-level PLL in your lower-level project so that you can optimize the design with information about the frequency multipliers, phase shifts, compensation delays, and any other PLL parameters. The software then captures timing and resource requirements more accurately while ensuring that the timing analysis in the lower-level project is complete and accurate. You can export the lower-level partition, without exporting any auxiliary components to the top-level design.

In addition, you can use this feature in a top-down design flow to create QXP files for specific design partitions that are complete. You can then import the QXP file back into the project and use the **Imported** netlist type. In this usage, the QXP file acts as an archive for the partition, including the netlist and placement and routing information in one file. If you need to change the source code for the partition, you must change the netlist type back to **Source File** to use the source instead of the imported information.

Importing a Lower-Level Partition Into the Top-Level Project

The import process involves importing the design netlist from the Quartus II Exported Partition file and adding the netlist to the database for the top-level project. Importing also filters the assignments from the subdesign and creates the appropriate assignments in the top-level project.

To import a subdesign partition into a top-level design:

1. In the top-level project, on the Project menu, click **Import Design Partition**. Alternatively, right-click on the partition that you want to import in the Design Partitions window and click **Import Design Partition**. The **Import Design Partition** dialog box appears.

2. In the **Partition(s)** box, browse to the desired partition. To choose a partition, highlight the partition name in the **Select Partition(s)** dialog box and use the appropriate buttons to select or deselect the desired partitions.

   Note that you can select multiple partitions if your top-level design has multiple instances of the subdesign partition and you want to use the same imported netlist.

3. Under **Import file**, type the name of the Quartus II Exported Partition file or browse for the file that you want to import into the selected partition. Note that this file is required only during importation, and is not used during subsequent compilations unless you reimport the partition.

   If you have already imported the Quartus II Exported Partition file for this partition at least once, you can use the same location as the previous import instead of specifying the file name again. To do so, turn on **Reimport using the latest import files at previous locations**. This option is especially useful when you want to import the new Quartus II Exported Partition files for several partitions that you have already imported at least once. You can select all the partitions to be imported in the Partition(s) box and then use the **Reimport using latest import files at previous locations** option to import all partitions using their previous locations, without specifying individual file names.

4. To view the contents of the selected Quartus II Exported Partition file, click **Load Properties**. The properties displayed include the **Netlist Type**, **Entity name**, **Device**, and statistics about the partition size and ports.

5. Click **Advanced Import Settings** and make selections, as appropriate, to control how assignments and regions are integrated from a subdesign into a top-level design partition. During importation, some regions may be resized or slightly moved. Click **OK** to apply the settings.

   For more information about the advanced settings, refer to "Importing Assignments and Advanced Import Settings" on page 2–37.

6. In the **Import Design Partition** dialog box, click **OK** to start importation. The specified Quartus II Exported Partition file is imported into the database for the current top-level project.

Importing Assignments and Advanced Import Settings

When you import a subdesign partition into a top-level design, the software sets certain assignments by default and also imports relevant assignments from the subdesign into the top-level design.

Design Partition Properties after Importing

When you import a subdesign partition, the import process sets the partition's Netlist Type to **Imported**.

If you compile the design and make changes to the place-and-route results, use the **Post-Fit (Import-based)** Netlist Type on the subsequent compilation. To discard an imported netlist and recompile from source code, simply compile the partition with netlist type set to **Source File** and be sure to include the relevant source code with the top-level project.

The import process sets the partition's Fitter Preservation Level to the setting with the highest degree of preservation supported by the imported netlist. For example, if a post-fit netlist is imported with placement information, the level is set to **Placement**, but you can change it to the **Netlist Only** value.

Refer to "Setting the Netlist Type for Design Partitions" on page 2–22 for details about the Netlist Type and Fitter Preservation Level setting.

Importing Design Partition Assignments Within the Subdesign

Design partition assignments defined within the subdesign project are not imported into the top-level project. All logic in the subdesign is imported as one partition in the QXP file.

Synopsys Design Constraint (SDC) Files for the Quartus II TimeQuest Timing Analyzer

Timing assignments made for the Quartus II TimeQuest Timing Analyzer in an SDC file are currently not imported into the top-level project. You should manually ensure that the top-level project includes all of the timing requirements for the entire project.

If you want to copy lower-level SDC files to the top-level project, consider prefixing lower-level constraints with a variable for the design hierarchy. Then, when you copy the file to the top-level design, you can set the variable to provide the hierarchy path to the lower-level partition in the top-level design.

Importing LogicLock Assignments

LogicLock regions are set to a fixed size when imported. If you instantiate multiple instances of a subdesign in the top-level design, the imported LogicLock regions are set to a Floating location. Otherwise, they are set to a Fixed location. You can change the location of LogicLock regions after they are imported, or change them to a Floating location to allow the software to place each region but keep the relative locations of nodes within the region wherever possible. If you want to preserve changes made to a partition after compilation, use the Netlist Type **Post-Fit (Import-Based)**.

The LogicLock Member State assignment is set to **Locked** to signify that it is a preserved region.

LogicLock back-annotation and node location data is not imported because the Quartus II Exported Partition file contains all the relevant placement information. Altera strongly recommends that you do not add to or delete members from an imported LogicLock region.

Importing Other Instance Assignments

All instance assignments are imported, with the exception of design partition assignments, SDC constraints, and LogicLock assignments, as described previously.

Importing Global Assignments

Global assignments are not imported. The project lead should make global assignments in the top-level design. Global assignments include clock settings for the Quartus II Classic Timing Analyzer.

Advanced Import Settings

The **Advanced Import Settings** dialog box allows you to specify the options that control how assignments and regions are integrated and how to resolve assignment conflicts when importing a subdesign partition into a top-level design. The following subsections describe each of these options.

Allow Creation of New Assignments
Allows the import command to add new assignments from the imported project to the top-level project.

When this option is turned off, it imports updates to existing assignments, but no new assignments are allowed.

Promote Assignments to all Instances of the Imported Entity
Converts and promotes entity-level assignments from the subdesign into instance-level assignments in the top-level design.

Assignment Conflict Resolution: LogicLock Regions
Choose one of the following options to determine how to handle conflicting LogicLock assignments (that is, subdesign assignments that do not match the top-level assignments):

- **Always replace regions in the current project** (default)—Deletes existing regions and replaces them with the new subdesign region. Any changes made to the LogicLock region after the assignments were imported are also deleted.
- **Always update regions in the current projects**—Overwrites existing region assignments to reflect any new subdesign assignments with the exception of the LogicLock Origin, in case the project lead has made floorplan location assignments in the top-level design.
- **Skip conflicting regions**—Ignores and does not import subdesign assignments that conflict with any assignments that exist in the top-level design.

Assignment Conflict Resolution: Other Assignments
Choose one of the following options to determine how to handle conflicts with other types of assignments (that is, the subdesign assignments do not match the top-level assignments):

- **Always replace assignments in the current project** (default)—Overwrites or updates existing instance assignments with the new subdesign assignments.
- **Skip conflicting assignments**—Ignores and does not import subdesign assignments that conflict with any assignments that exist in the top-level design.

Generating Bottom-Up Design Partition Scripts for Project Management

The bottom-up design partition scripts automate the process of transferring top-level project information to lower-level modules. The software provides a project manager interface for managing resource and timing budgets in the top-level design. This makes it easier for designers of lower-level modules to implement the instructions from the project lead, and avoid conflicts between projects when importing and incorporating the projects into the top-level design. This helps reduce the need to further optimize the designs after integration, and improves overall designer productivity and team collaboration.



Generating bottom-up design partition scripts is optional in any bottom-up design methodology.

For example design scenarios using these scripts, refer to "Bottom-Up Incremental Design Flows" on page 2–67. In a typical bottom-up design flow, the project lead must perform some or all of the following tasks to ensure successful integration of the subprojects:

- Manually determine which assignments should be propagated from the top level to the bottom levels. This requires detailed knowledge of which Quartus II assignments are needed to set up low-level projects.
- Manually communicate the top-level assignments to the low-level projects. This requires detailed knowledge of Tcl or other scripting languages to efficiently communicate project constraints.
- Manually determine appropriate timing and location assignments that will help overcome the limitations of bottom-up design. This requires examination of the logic in the lower levels to determine appropriate timing constraints.
- Perform final timing closure and resource conflict avoidance at the top level. Because the low-level projects have no information about each other, meeting constraints at the lower levels does not guarantee they will be met when integrated at the top-level. It then becomes the project lead's responsibility to resolve the issues, even though information about the low-level implementation may not be available.

Using the Quartus II software to generate bottom-up design partition scripts from the top level of the design makes these tasks much easier and eliminates the chance of error when communicating between the project lead and lower-level designers. Partition scripts pass on assignments made in the top-level design, and create some new assignments that guide the placement and help the lower-level designers see how their design connects to other partitions. If necessary, you can exclude specific design partitions.

Generate design partition scripts after a successful compilation of the top-level design. On the Project menu, click **Generate Bottom-Up Design Partition Scripts**. The design can have empty partitions as placeholders for lower-level blocks, and you can perform an Early Timing Estimation instead of a full compilation to reduce compilation times.

The following subsections describe the information that can be included in the bottom-up design partition Tcl scripts. Use the options in the **Generate Bottom-Up Design Partition Scripts** dialog box to choose which types of assignments you want to pass down and create in the lower-level partition projects. Each time you rerun the script generation process, the Quartus II software recreates the files and replaces older versions.

For information about current limitations in the bottom-up partition scripts, refer to "Bottom-Up Design Partition Script Limitations" on page 2–84.

Project Creation

You can use the **Create lower-level project if one does not exist** option for the partition scripts to create lower-level projects if they are required. The Quartus II Project File for each lower-level project has the same name as the entity name of its corresponding design partition.

With this project creation feature, the scripts work by themselves to create a new project, or can be sourced to make assignments in an existing project.

Excluded Partitions

Use the **Excluded partition(s)** option at the bottom of the dialog box to exclude specific partitions from the Tcl script generation process. Use the browse button, then highlight the partition name in the **Select Partition(s)** dialog box and use the appropriate buttons to select or deselect the desired partitions.

Assignments from the Top-Level Design

By default, any assignments made at the top level (not including default assignments or project information assignments) are passed down to the appropriate lower-level projects in the scripts. The software uses the assignment variables and determines the logical partition(s) to which the assignment pertains (this includes global assignments, instance assignments, and entity-level assignments). The software then changes the assignments so that they are syntactically valid in a project with its target partition's logic as the top-level entity.

The names of the design files that apply to the specific partition are added to each lower-level project. Note that the script uses the file name(s) specified in the top-level project. If the top-level project used a placeholder wrapper file with a different name than the design file in the lower-level project, you should be sure to add the appropriate file to the lower-level project.

The scripts process wildcard assignments correctly, provided there is only one wildcard. Assignments with more than one wildcard are ignored and warning messages are issued.

Use the following options to specify which types of assignments to pass down to the lower-level projects:

- **Timing assignments**—When this option is turned on, all Classic Timing Analyzer global timing assignments for the lower-level projects are included in the script, including t_{CO}, t_{SU}, and f_{MAX} constraints. This option may also include timing constraints on internal partition connections.
- **Design partition assignments**—When this option is turned on, script assignments related to design partitions in the lower-level projects are included, as well as assignments associated with LogicLock regions.
- **Pin location assignments**—When this option is turned on, all pin location assignments for lower-level project ports that connect to pins in the top-level design are included in the script, controlling the overuse of I/Os at the top-level during the integration phase and preserving placement.

Virtual Pin Assignments

When **Create virtual pins at low-level ports connected to other design units** is turned on, the Quartus II software searches partition netlists and identifies all ports that have cross-partition dependencies. For each lower-level project pin associated with an internal port in another partition or in the top-level project, the script generates a virtual pin assignment, ensuring more accurate placement, because virtual pins are not directly connected to I/O ports in the top-level project. These pins are removed from a lower-level netlist when it is imported into the top-level design.

Virtual Pin Timing and Location Assignments

One of the main issues in bottom-up design methodologies is that each individual design block includes no information about how it is connected to other design blocks. If you turn on the option to write virtual pin assignments, you can also turn on options to constrain these virtual pins to achieve better timing performance after the lower-level partitions are integrated at the top level.

When **Place created virtual pins at location of top-level source/sink** is turned on, the script includes location constraints for each virtual pin created. Virtual output pins are assigned to the location of the connection's destination in the top-level project, and virtual input pins are assigned to the location of the connection's source in the top-level project. Note that if the top-level design uses Empty partitions, the final location of the connection is not known but the pin is still assigned to the LogicLock region that contains its source or destination.

As a result, these virtual pins are no longer placed inside the LogicLock region of the lower-level project, but at their location in the top-level design, eliminating resource consumption in the lower-level project and providing more information about lower-level projects and their port dependencies. These location constraints are not imported into the top-level project.

When **Add maximum delay to created virtual input pins**, **Add maximum delay from created virtual output pins**, or both, are turned on, the script includes timing constraints for each virtual pin created. The value you enter in the dialog box is the maximum delay allowed to or from all paths between virtual pins to help meet the timing requirements for the complete design. The software uses the `INPUT_MAX_DELAY` assignment or `OUTPUT_MAX_DELAY` assignment to apply the constraint.

This option allows the project lead to specify a general timing budget for all lower-level internal pin connections. The lower-level designer can override these constraints by applying individual node-level assignments on any specific pin as needed.

LogicLock Region Assignments

When **Copy LogicLock region assignments from top-level** is turned on, the script includes assignments identifying the LogicLock assignment for the partition.

The script can also pass assignments to create the LogicLock regions for all other partitions. When **Include all LogicLock regions in lower-level projects** is turned on, the script for each partition includes all LogicLock region assignments for the top-level project and each lower-level partition, revealing the floorplan for the complete design in each partition. Regions that do not belong to other partitions contain virtual pins representing the source and destination ports for cross-partition connections. This allows each designer to more easily view the connectivity between their partition and other partitions in the top-level design, and helps ensure that resource conflicts at the top level are minimized.

When **Remove existing LogicLock regions from lower-level projects** is turned on, the script includes commands to remove LogicLock regions defined in the lower-level project prior to running the script. This ensures that LogicLock regions not part of the top-level project do not become part of the complete design, and avoids any location conflicts by ensuring lower-level designs use the LogicLock regions specified at the top level.

Global Signal Promotion Assignments

To help prevent conflicts in global signal usage when importing projects into the top-level design, you can choose to write assignments that control how signals are promoted to global routing resources in the lower-level partitions. These options can help resource balancing of global routing resources.

When **Promote top-level global signals in lower-level projects** is turned on, the Quartus II software searches partition netlists and identifies global resources, including clock signals. For the relevant partitions, the script then includes a global signal promotion assignment, providing information to the lower-level projects about global resource allocation.

When **Disable automatic global promotion in lower-level projects** is turned on, the script includes assignments that turn off all automatic global promotion settings in the lower-level projects. These settings

include the Auto Global Memory Control Signals logic option, output enable logic options, and clock and register control promotions. If you select the **Disable automatic global promotion in lower-level projects** option in conjunction with the **Promote top-level global signals in lower-level projects** option, you can ensure that only signals promoted to global resources in the top-level are promoted in the lower-level projects.

Makefile Generation

Makefiles allow you to use **make** commands to ensure that a bottom-up project is up-to-date if you have a make utility installed on your computer. The **Generate makefiles to maintain lower-level and top-level projects** option creates a makefile for each design partition in the top-level design, as well as a master makefile that can run the lower-level project makefiles. The Quartus II software places the master makefiles in the top-level directory, and the partition makefiles in their corresponding lower-level project directories.

You must specify the dependencies in the makefiles to indicate which source file should be associated with which partition. The makefiles use the directory locations generated using the **Create lower-level project if one does not exist** option. If you created your lower-level projects without using this option, you must modify the variables at the top of the makefile to specify the directory location for each lower-level project.

To run the makefiles, use a command such as **make -f master_makefile.mak** from the script output directory. The master makefile first runs each lower-level makefile, which sources its Tcl script and then generates a Quartus II Exported Partition file to export the project as a design partition. Next, run the top-level makefile that specifies these newly generated Quartus II Exported Partition files as the import files for their respective partitions in the top-level project. The top-level makefile then imports the lower-level results and performs a full compilation, producing a final design.

To exclude a certain partition from being compiled, edit the `EXCLUDE_FLAGS` section of **master_makefile.mak** according to the instructions in the file, and specify the appropriate options. You can also exclude some partitions from being built, exported, or imported using **make** commands. To exclude a partition, run the makefile using a command such as the one for the GNU **make** utility shown in the following example:

```
gnumake –f master_makefile.mak exclude_<partition directory>=1 ↵
```

This command instructs that the partition whose output files are in <*partition directory*> are not built. Multiple directories can be excluded by adding multiple `exclude_`<*partition directory*> commands. Command-line options override any options in the makefile.

Another feature of makefiles is the ability to have the master makefile invoke the low-level makefiles in parallel on systems with multiple processors. This option can help designers working with multiple CPUs greatly improve their compilation time. For the GNU make utility, add the `-j`<*N*> flag to the **make** command. The value <*N*> is the number of processors that can be used to run the build.



The makefile does not include a make clean option, so the design may recompile when **make** is run again and a QXP file already exists.

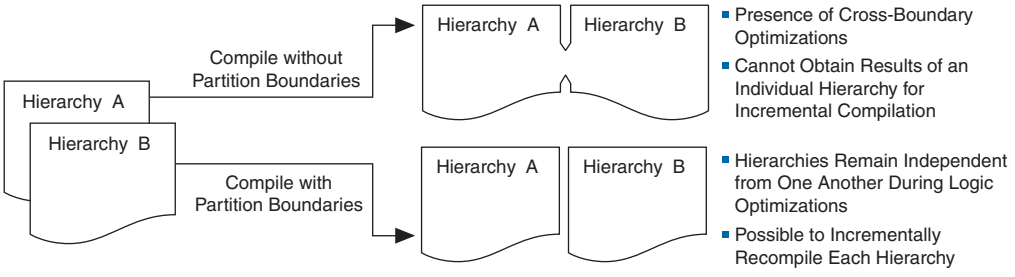
Guidelines for Creating Good Design Partitions and LogicLock Regions

This section provides guidelines for creating design partitions and floorplan location assignments that will help you achieve good quality results, as well as criteria and methodologies to check the quality of your assignments.

When planning your design, keep in mind the size and scope of each partition, and the likelihood that different parts of your design might change as your design develops. Consider placing logic that changes frequently into its own partition, so that you have to recompile only that partition if the rest of the design stays the same. Similarly, consider placing fixed logic, such as IP cores or logic reused from another project, into its own partition so that you can compile once and lock down the placement immediately with a post-fit netlist.

Creating partitions prevents the compiler from performing logic optimizations across partition boundaries (Figure 2–10), allowing the software to synthesize and place each partition separately.

Figure 2–10. Effects of Partition Boundaries During Optimization



For example, consider a design with a 36-bit function defined in partition A, but with only 18 bits connected in partition B. In a flat design, you would expect the logic for the other 18 bits to be removed during synthesis. With incremental compilation, the Quartus II compiler does not remove the (unused) logic from partition A because partition B is allowed to change independently from partition A. That is, you could later connect all 36 bits in partition B. In this case, you should remove the unconnected ports and replace them with ground signals inside partition A. You can create a new wrapper file to do this.

Another example is the case in which a clock is inverted at partition boundary, but the inversion should be done in the destination LAB for best results. With incremental compilation, the Quartus II compiler uses logic to invert the signal, then routes the signal on global clock resource to its destinations within the partition. The signal acts as a gated clock with high skew. You must set up partitions to ensure that optimization does not rely on information from other partitions, so you should perform clock inversions in the destination partitions.

Because cross-boundary optimizations cannot occur when using partitions, the quality of results and performance of the design may decrease as the number of partitions increases. Although more partitions allows for greater reduction in compilation time, you should limit the number of partitions to prevent degradation of the quality of results. This effect is more pronounced in a bottom-up methodology than a top-down methodology.

In a top-down compilation where partitions are not locked down with post-fitting results, the Fitter can perform placement optimizations on the design as a whole to optimize the placement of cross-partition paths. (However, the Fitter cannot perform logic optimizations such as physical synthesis across the partition boundary.) In a bottom-up flow, partitions are compiled separately. Typically, the fitting results are exported, so there is no placement optimization across the partitions boundaries.

Creating Good Design Partitions

Altera recommends that you observe the following important hierarchical design considerations when creating partitions:

- Register all inputs and outputs of each partition. This helps avoid any delay penalty on signals that cross partition boundaries. At the very least, either the inputs or the outputs should be registered. The Statistics reports described in the "Partition Statistics Reports" section list the ports registered for each partition.



While this can be difficult in practice, adherence to this principle results in less timing degradation and area increase when using incremental flows. Registering lessens the need for the cross-partition optimizations that are prevented by partitioning. By registering the ports, you can keep critical paths within a single partition, thus keeping the lengths of inter-partition register-to-register paths to a minimum.

- Minimize the number of paths that cross partition boundaries. If there are critical paths crossing between partitions, rework the partitions to avoid these inter-partition paths. Capturing as many of the timing-critical connections as possible inside a partition allows you to effectively apply optimizations to that partition to improve timing, while leaving the rest of the design unchanged. The Statistics reports described in "Partition Statistics Reports" on page 2–50 list the number of input and output ports for each partition.
- Ensure that the size of each partition is not too small (as a rough guideline, partitions should be greater than 2,000 logic elements (LEs) or adaptive logic modules (ALMs)). The Statistics reports described in the "Partition Statistics Reports" section list the logic utilization of each partition.
- Minimize the number of unconnected ports at partition boundaries. When a port is left unconnected, optimizations that remove logic driving that port could improve results. However, these optimizations are not allowed in an incremental design, because they would lead to cross-partition dependence. Altera recommends that you either connect such ports to an appropriate node or remove them from the design. If you know the port should not be used, consider defining a wrapper module with a port interface that reflects this fact. The Statistics reports described in the "Partition Statistics Reports" section list the number of unconnected input and output ports for each partition.

- Do not use tri-state signals or bidirectional ports on hierarchical boundaries, unless the port is connected directly to a top-level I/O pin on the device. If you use boundary tri-states in a lower-level block, synthesis pushes the tri-states through the hierarchy to the top level to take advantage of the tri-state drivers on the output pins of the device.
  In an incremental compilation flow, internal tri-states are supported only when all the destination logic is contained in the same partition, in which case Analysis and Synthesis implements the internal tri-state signals using multiplexing logic. For a bidirectional port that feeds a bidirectional pin at the top level, all the logic that forms the bidirectional I/O cell must reside in the same partition.
- Note that logic is not synthesized or optimized across partition boundaries, which means any constant value (for example, a signal set to `GND`) is not propagated across partitions. If a port is supposed to be connected to `VCC` or `GND`, replace the port with `VCC` or `GND` in the module's design. This allows optimizations to take place that could not be performed if `VCC` or `GND` is connected through a port.
- Do not use the same signal to drive multiple ports on a single partition. If the same driving signal feeds multiple ports of a partition, those ports are logically equivalent. However, because inter-partition optimizations cannot be performed, the compilation of that partition cannot take advantage of this fact, which usually results in sub-optimal performance. For example, if a single clock is used to drive the read and write clocks of a RAM block and the RAM block is compiled separately in a bottom-up design flow, the RAM block is implemented as though there are two unique clocks. If you know the port connectivity will not change (that is, the ports will always be driven by the same signal), redefine the port interface so there is only a single port that can then internally drive other logic in the partition. If required, you can create a wrapper module around the partition that has fewer ports.
- Do not directly connect two ports of a partition. If two ports on a module are directly connected, consider redefining the module to remove those ports. If an output port drives an input port on the same module, the connection can be made internally without going through any I/O ports. If an input port drives an output port directly, the connection can likely be implemented without the ports by connecting the signals in a higher-level design partition.
- You may have to perform some manual resource balancing across partitions if device resources are overused in the individual partitions. Refer to "Resource Balancing" on page 2–51 for details.
- You may have to perform some timing budgeting if paths that cross partition boundaries require further optimization. Refer to "Timing Budgeting" on page 2–53 for details.

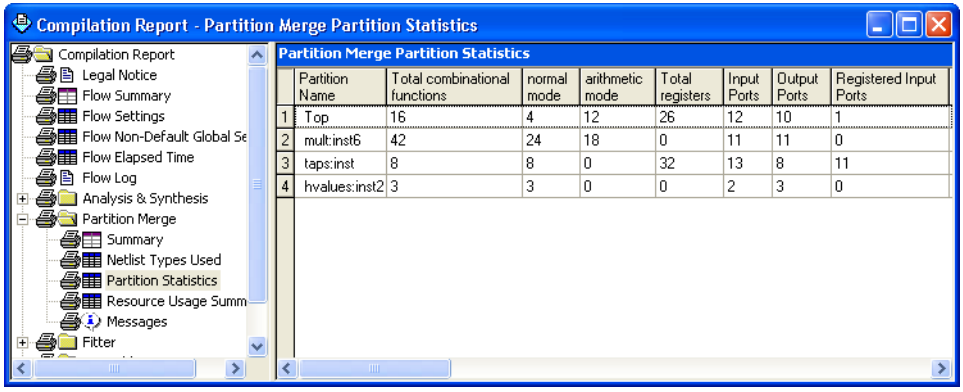
You can use the Incremental Compilation Advisor to check that your design follows many of these guidelines. Refer to "Incremental Compilation Advisor" on page 2–60 for more details.

Partition Statistics Reports

You can view statistics about design partitions in the Partition Merge Partition Statistics compilation report and the **Statistics** tab in the **Design Partitions Properties** dialog box.

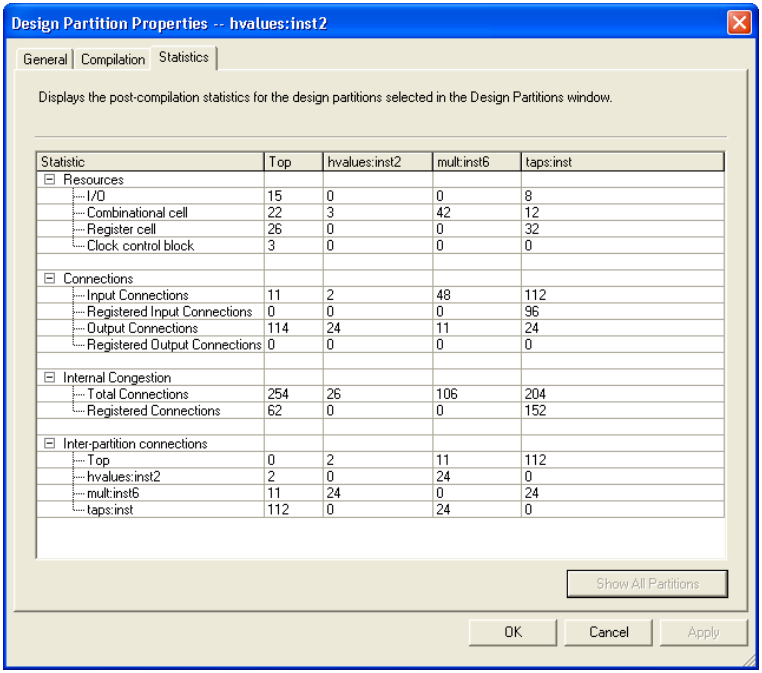
The **Partition Statistics** page under the **Partition Merge** folder of the **Compilation Report** lists statistics about each partition. The statistics for each partition (each row in the table) include the number of logic cells it contains, as well as the number of input and output pins it contains and how many are registered or unconnected. This report is useful when optimizing your design partitions in a top-down compilation flow, or when you are compiling the top-level design in a bottom-up compilation flow, ensuring that the partitions meet the guidelines presented in "Creating Good Design Partitions" on page 2–47. Figure 2–11 shows the report window.

Figure 2–11. Partition Merge Partition Statistics Report



You can also view statistics about the resource and port connections for a particular partition on the **Statistics** tab of the **Design Partition Properties** dialog box. On the Assignments menu, click **Design Partitions Window**. Right-click on a partition and click **Properties** to open the dialog box. Click **Show All Partitions** to view all the partitions in the same report (Figure 2–12).

Figure 2–12. Statistics Tab in the Design Partitions Properties Dialog Box



Statistic	Top	hvalues:inst2	mult:inst6	taps:inst
⊟ Resources				
I/O	15	0	0	8
Combinational cell	22	3	42	12
Register cell	26	0	0	32
Clock control block	3	0	0	0
⊟ Connections				
Input Connections	11	2	48	112
Registered Input Connections	0	0	0	96
Output Connections	114	24	11	24
Registered Output Connections	0	0	0	0
⊟ Internal Congestion				
Total Connections	254	26	106	204
Registered Connections	62	0	0	152
⊟ Inter-partition connections				
Top	0	2	11	112
hvalues:inst2	2	0	24	0
mult:inst6	11	24	0	24
taps:inst	112	0	24	0

Resource Balancing

When using incremental compilation, the software synthesizes each partition separately, with no data about the resources used in other partitions. This means that device resources could be overused in the individual partitions during synthesis, and thus the design may not fit in the target device when the partitions are merged.

In a bottom-up design flow in which designers optimize their lower-level designs and export them to a top-level design, the software also places and routes each partition separately. In some cases, partitions can use conflicting resources when combined at the top level.

To avoid these effects, you may have to perform manual resource balancing across partitions.

RAM and DSP Blocks

In the standard synthesis flow, when DSP blocks or RAM blocks are overused, the Quartus II Compiler can perform resource balancing and convert some of the logic into regular logic cells (for example, LEs or ALMs). Without data about resources used in other partitions, it is possible for the logic in each separate partition to maximize the use of a particular device resource, such that the design does not fit after all the partitions are merged. In this case, you may be able to manually balance the resources by using the Quartus II synthesis options to control inference of megafunctions that use the DSP or RAM blocks. You can also use the MegaWizard® Plug-In Manager to customize your RAM or DSP megafunctions to use regular logic instead of the dedicated hardware blocks.



For more information about resource balancing when using Quartus II synthesis, refer to the *Megafunction Inference Control* section in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For more tips about resource balancing and reducing resource utilization, refer to the appropriate *Resource Utilization Optimization Techniques* section in the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Altera recommends using a LogicLock region for each partition to minimize the chance that the logic in more than one partition uses the same logic resource. However, there are situations in which partition placement may still cause conflicts at the top level. For example, you can design a partition one way in a lower-level design (such as using an M-RAM memory block) and then instantiate it in two different ways in the top level (such as one using an M-RAM block and another using an M4K block). In this case, you can use a post-fit netlist only with no placement information to allow the software to refit the logic.

Global Routing Signals

Global routing signals can cause conflicts when multiple projects are imported into a top-level design. The Quartus II software automatically promotes high fan-out signals to use global routing resources available in the device. Lower-level partitions can use the same global routing resources, thus causing conflicts at the top level.

In addition, LAB placement depends on whether the inputs to the LCELLs within the LAB are using a global clock signal. Therefore, problems can occur if a design does not use a global signal in the lower-level design, but does use a global signal in the top-level design.

To avoid these problems, the project lead can first determine which partitions will use global routing signals. Each designer of a lower-level partition can then assign the appropriate type of global signals manually, and prevent other signals from using global routing resources. If you have all partitions available, you can compile the entire design at the top level with floorplan assignments to allow the use of regional clocks that span only a part of the chip. The Fitter automatically promotes some signals to global routing, and you can use this information when optimizing the lower-level partitions in separate Quartus II projects.

Use the **Global Signal** assignment set to a value of **On** or **Off** in the Assignment Editor to place a signal on a global routing line, or to prevent the signal from using a global routing line. You can also assign certain types of global clock resources in some device families, such as regional clocks that cover only part of the device. You can view the resource coverage of such clocks in the Chip Planner, and then align LogicLock regions that constrain partition placement with available global clock routing resources. For example, if the LogicLock region for a particular partition is limited to one device quadrant, that partition's clock can use a regional clock routing type that covers only one device quadrant.

If you want to disable the automatic global promotion performed in the Fitter, turn off the **Auto Global Clock** and **Auto Global Register Control Signals** options. On the Assignments menu, click **Settings**. On the **Fitter Settings** page, click **More Settings** and change the settings to **Off**.

Alternatively, to avoid problems when importing, direct the Fitter to discard the placement and routing of the imported netlist by setting the Fitter preservation level property of the partition to **Netlist Only**. With this option, the Fitter re-assigns all the global signals for this particular partition when compiling the top-level design.

If you are performing a bottom-up flow using the design partition scripts, the software can automatically write the commands to pass global constraints and turn off the automatic options. Refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–40 for details.

Timing Budgeting

If you optimize lower-level partitions independently and import them to the top level, any unregistered paths that cross between partitions are not optimized as an entire path. One way to reduce this effect is to ensure input and output ports of the partitions are registered whenever possible.

To ensure that the Compiler correctly optimizes the input and output logic in each partition, you may be required to perform some manual timing budgeting. For each unregistered timing path that crosses between partitions, make timing assignments on the corresponding I/O path in each partition to constrain both ends of the path to the budgeted timing delay. Timing budgets may be required for these I/O ports because when the Compiler optimizes each partition, it has no information about the placement of the logic that connects to that port. If the logic in one partition is placed far away from logic in another partition, the routing delay between the logic could lead to problems meeting the timing requirements. Assigning a timing budget for each part of the connection ensures that the Compiler optimizes the paths appropriately.

When performing manual timing budgeting, you can also use **Virtual Pin** assignments to represent I/O ports of a partition that feed another partition in the full design. By assigning location and timing constraints to the **Virtual Pins** that represent the connections in the full design, you can further improve the quality of the timing budget.

If you are performing a bottom-up flow using the design partition scripts, the software can write virtual pin assignments and I/O timing budget constraints automatically. Refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–40 for details.

Methodology to Check Partition Quality during Partition Planning

There is an inherent tradeoff between compilation time and quality of results when you vary the number of partitions in a project. You can ensure that you limit this effect by following an iterative methodology during the partitioning process. In any incremental compilation flow in which you can compile the source code for each partition during the partition planning phase, Altera recommends the following iterative flow:

1. Start with a complete design that is not partitioned and has no location or LogicLock assignments.

2. To perform a placement and timing analysis estimate, on the Processing menu, point to Start and click **Start Early Timing Estimate**.



You must perform Analysis and Synthesis before performing an Early Timing Estimate. If incremental compilation is already turned on, you must also perform Partition Merge.

To run a full compilation instead of the Early Timing Estimate, on the Processing menu, click **Start Compilation**.

3. Record the quality of results from the Compilation Report (f_{MAX}, area, and so forth).

4. Create design partitions as described in "Creating Design Partitions" on page 2–19 using the guidelines in "Creating Good Design Partitions" on page 2–47.

5. Perform another Early Timing Estimate or full compilation.

6. Record the quality of results from the Compilation Report. If the quality of results is significantly worse than that obtained in the previous compilation in Step 3, repeat Step 4 through this step (Step 6) to change your partition assignments and use a different partitioning scheme.

7. Even if the quality of results is acceptable, you can repeat Step 4 through Step 6 by further dividing a large partition into several smaller partitions. Doing so improves compilation time in future incremental compilations. You can repeat this step until you achieve a good tradeoff point (that is, all critical paths are localized within partitions, the quality of results is not negatively affected, and the size of each partition is reasonable).

The Importance of Floorplan Location Assignments in Incremental Compilation

Floorplan location planning can be very important for a design that uses full incremental compilation, for the following two reasons:

- To avoid resource conflicts between partitions
- To ensure a good quality of results when recompiling partitions and other partition placement is unchanged

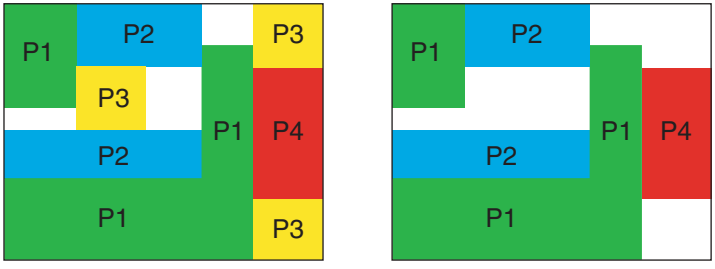
Location assignments for each partition ensures that there are no conflicts for locations between different partitions. If there are no LogicLock region assignments, or if LogicLock regions are set to auto-size or floating, it is unclear which resources on the device are allocated for the logic associated with the region. Without clearly defining this resource budget, bottom-up design can produce many resource conflicts when

importing results, because each bottom-up partition has no information about its resource budget and may therefore claim resources required by another partition.

In addition, a design floorplan helps to avoid the situation that arises when the Fitter is directed to place or replace a portion of the design in an area of the device where most resources have already been claimed. In this case, the placement of the post-fit netlists of other modules forces the Fitter to place the new portion of the design in the empty parts of the device. There are two immediate disadvantages to this situation. First, the Fitter must work harder because of the higher number of physical constraints, and therefore compilation time probably increases. Second, the quality of results often decreases, sometimes dramatically, because the placement of the target module is now scattered throughout the device.

Figures 2–13 and 2–14 illustrate the problems associated with refitting designs that do not have floorplan location assignments. Figure 2–13 shows the initial placement of a four-partition design (**P1**–**P4**) without floorplan location assignments. The second part of the figure shows the situation if a change occurs to **P3**. After removing the logic for the changed partition, the Fitter must replace and reroute the new logic for **P3** using the white space shown in the figure.

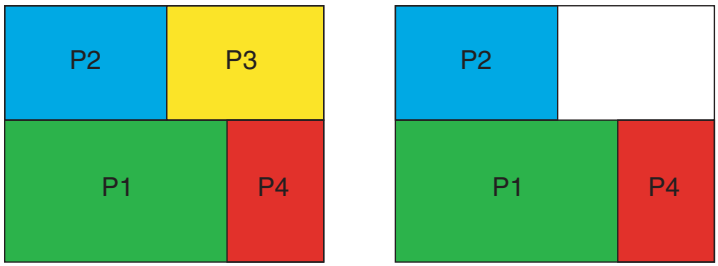
Figure 2–13. Representation of Device Floorplan without Location Assignments



Performing this placement is very difficult. The Fitter may not be able to find any legal placement for the logic in partition **P3**, even if it was able to do so in the initial compilation. If the Fitter does find a legal placement, the results are probably sub-optimal.

Figure 2–14 shows the initial placement of a four-partition design with floorplan location assignments made by the user, and the situation after partition **P3** is removed in this case.

Figure 2–14. Representation of Device Floorplan with Location Assignments



This placement presents a much more reasonable task to the Fitter and yields better results than the previous case that does not have floorplan location assignments. Due to this effect, you should ensure that you have a LogicLock floorplan assignment for any timing-critical blocks that will be recompiled as you make changes to the design. You can use the Reserved property to ensure that there are no placement conflicts in bottom-up flows. Logic that is not timing-critical can float throughout the device in a top-down compilation flow, so a floorplan assignment might not be required in this case.

Creating Good Floorplan Location Assignments

This section presents recommendations for creating a design floorplan using LogicLock regions.

In most cases, each LogicLock region should contain logic from only one partition. This organization helps prevent resource conflicts in a bottom-up design and can lead to better performance preservation when locking down parts of a project in a top-down design. One exception to this rule is the case where you want to have two lower-level partitions compiled together in the same LogicLock region because of tight interaction, but you want to separate the placement of the parent logic for each partition. In this case, you can place more than one partition in one LogicLock region, but for best results you must ensure that you recompile all partitions every time the logic in one partition changes. In addition, if your partition consists of a wrapper around more than one lower-level

module, you can place those modules in different areas of the device by using different LogicLock regions even if they are defined in the same partition.

If your design contains hierarchical partitions (that is, parent-child relationships between partitions), you can create hierarchical LogicLock regions to ensure that the logic in the child partition is physically placed inside the LogicLock region for the parent partition. This can be useful when the parent partition does not contain registers at the boundary with the lower-level child partition and has a lot of signal connectivity. To create a hierarchical relationship between regions in the **LogicLock Regions** window, drag and drop the child region to the parent region.

Ensure that all LogicLock regions in the design have a fixed size and have their origin locked to a specific location on the chip. If you use auto-sized, floating-location regions to create an initial floorplan, be sure to set the size and origin to use the fitter results before you recompile. Do not use the Soft LogicLock region property. Refer to "The Importance of Floorplan Location Assignments in Incremental Compilation" on page 2–55 for more information.

If resource utilization is low, you can enlarge the regions chosen by the Fitter with the auto-size setting. Doing so usually improves the final results because it gives the Fitter more freedom to place additional logic added to the partition during future incremental compilations.

Ideally, almost the entire device should be covered by LogicLock regions if all partitions are assigned to a region. Give more area to regions that are densely populated, because overly congested regions can lead to poor results. You may move the region origins from auto-floating region placement to satisfy this requirement, but Altera recommends preserving the Fitter-determined relative placement of the regions. Also, regions that are too large for their logic can result in wasted resources and also lead to poor results. You should define LogicLock regions that are neither too small nor too large.

Regions should not overlap in the device floorplan, especially in bottom-up flows. If two partitions are allocated an overlapping portion of the chip, each may independently claim some common resources in this region. This will lead to resource conflicts when importing bottom-up results into a final top-level design.

If two LogicLock regions have several connections between them, place them near each other to improve timing performance. By placing connected regions near each other, the Fitter has more opportunity to

optimize inter-region paths when both partitions are recompiled. Reducing the criticality of inter-region paths also allows the Fitter more flexibility when placing the other logic in each region.

You can use the Incremental Compilation Advisor to check that your design follows many of these guidelines. Refer to "Incremental Compilation Advisor" on page 2–60 for more details.



For more information about making and editing LogicLock regions, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Excluding Certain Device Elements (such as RAM or DSP Blocks) with Resource Exceptions

If your design contains memory or digital signal processing (DSP) elements, you may want to exclude these elements from the LogicLock region. You can use LogicLock resource exceptions to prevent elements of certain types from being assigned to a region. Note that the filter does not prevent them from being placed inside the region boundaries unless the region's **Reserved** property is turned on. Defining a resource exception instructs the Fitter that certain blocks are not required to be inside a region.

Resource exceptions are useful in cases where it is difficult to place rectangular regions for design blocks that contain memory and DSP elements, because of their placement in columns throughout the device floorplan. Excluding these elements can help to resolve no-fit errors that are caused by regions spanning too many resources, especially for designs that are memory-intensive, DSP-intensive, or both. If desired, you can also create separate regions for the memory or DSP blocks, excluding logic cell resources, which can be shaped to accommodate the columns in the device to control the placement of those design elements.

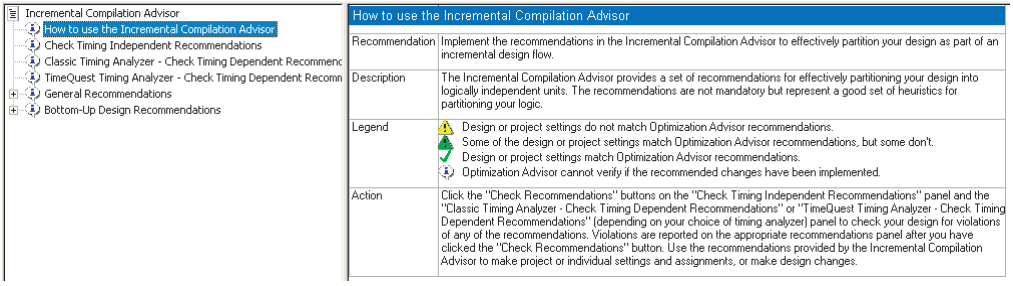
To view any resource exceptions, right-click in the LogicLock Regions window and click **Properties**. In the **LogicLock Region Properties** dialog box, highlight the design element (module/entity) in the **Members** box and click **Edit**. To set up a resource exception, click the browse button under **Excluded element types**, then turn on the design element types to be excluded from the region. You can choose to exclude combinational logic or registers from logic cells, or any of the sizes of TriMatrix™ memory blocks, or DSP blocks.

Incremental Compilation Advisor

You can use the Incremental Compilation Advisor to check that your design follows many of the recommendations presented in this chapter for creating design partitions and floorplan location assignments. On the Tools menu, point to Advisors, and click **Incremental Compilation Advisor**.

As shown in Figure 2–15, recommendations are split into **General Recommendations** that apply to all compilation flows and **Bottom-Up Design Recommendations** that apply to bottom-up design methodologies. Each recommendation provides an explanation, describes the effect of the recommendation, and provides the action required to make the suggested change. In some cases, there is a link to the appropriate Quartus II settings page where you can make a suggested change to assignments or settings.

Figure 2–15. Incremental Compilation Advisor



To check whether the design follows the recommendations, go to the **Timing Independent Recommendations** page or the **Timing Dependent Recommendations** page, and click **Check Recommendations**. For large designs, these operations can take a few minutes. After you perform a check operation, symbols appear next to each recommendation to indicate whether the design or project setting follows the recommendations, or if some or all of the design or project settings do not follow the recommendations. Refer to the Legend on the **How to use the Incremental Compilation Advisor** page in the advisor for more information.

For some items in the Advisor, if your design does not follow the recommendation, the **Check Recommendations** operation lists any parts of the design that could be improved. For example, if not all of the partition I/O ports follow the **Register All Ports** recommendation, the advisor displays a list of unregistered ports with the partition name and the source and destination nodes for the port.

When the advisor provides a list of nodes, you can right-click on a node and click **Locate** to cross-probe to other Quartus II features such as the RTL Viewer, Chip Planner, or the design source code in the text editor.



The first time you open the RTL or Technology Map Viewer, a preprocessor stage runs. This preprocessor resets the Incremental Compilation Advisor, so you must rerun the Check Recommendations process. Alternatively, you can open the appropriate netlist viewer before you use the Incremental Compilation Advisor if you want to locate nodes in the viewer.

Criteria for Successful Partition and Floorplan Schemes

The end results of design partitioning and floorplan creation differ from design to design. However, it is important to evaluate your results to ensure that your scheme is successful. Compare the results before creating your floorplan location assignments to the results after doing so, and consider using another scheme if any of the following guidelines are not met:

- No degradation in f_{MAX} should be observed after the design is partitioned and floorplan location assignments are created. In many cases, a slight increase in f_{MAX} is possible.
- The area increase should be no more than 5% after the design is partitioned and floorplan location assignments are created.
- The time spent in the routing stage should not significantly increase.

The amount of compilation time spent in the routing stage is reported in the Messages window with an Info message indicating the elapsed time for Fitter routing operations. If you notice a dramatic increase in routing time, the floorplan location assignments may be creating substantial routing congestion. In this case, decrease the number of LogicLock regions. Doing so typically reduces the compilation time in subsequent incremental compilations, and may also improve design performance. To help you modify your LogicLock regions, you can identify areas of congested routing in your design using the Chip Planner. On the Tools menu, click **Chip Planner**. To view the routing congestion, click the **Layers** icon located next to the Task menu. Under **Background Color Map**, select the **Routing Utilization map**.

Recommended Design Flows and Compilation Application Examples

This section provides design flows for solving common timing closure and team-based design issues using incremental compilation. Each flow describes the situation in which it should be used, and gives a step-by-step description of the commands required to implement the flow. These examples are divided into the following two sections:

- "Top-Down Incremental Design Flows"
- "Bottom-Up Incremental Design Flows"

Top-Down Incremental Design Flows

There are four top-down incremental design flow examples that reduce compilation time while making incremental changes to the design. The following design flow examples also allow you to achieve timing closure more quickly by optimizing or preserving the results for one partition in a larger design:

- "Design Flow 1—Changing a Source File for One of Multiple Partitions in a Top-Down Compilation Flow"
- "Design Flow 2—Optimizing the Placement for One of Multiple Partitions in a Top-Down Compilation Flow" on page 2–63
- "Design Flow 3—Preserving One Critical Partition in a Multiple-Partition Design in a Top-Down Compilation Flow" on page 2–64
- "Design Flow 4—Placing All but One Critical Partition in a Multiple-Partition Design in a Top-Down Compilation Flow" on page 2–65

All examples assume you have set up the project to use the full incremental compilation flow, using the steps described in "Quick Start Guide – Summary of Steps for an Incremental Compilation Flow" on page 2–11.

Design Flow 1—Changing a Source File for One of Multiple Partitions in a Top-Down Compilation Flow

Use this flow to update the source file in one partition without having to recompile the other parts of the design. You can reduce the compilation time by keeping the post-fit netlists for the unchanged partitions, while also preserving the performance for these blocks to reduce additional timing closure efforts.

Example background: You have just performed a lengthy, complete compilation of a design that consists of multiple partitions. An error is found in the HDL source file for one partition and it is being fixed. Because the design is currently meeting timing requirements and the fix is not expected to affect timing performance, it makes sense to compile only the affected partition and preserve the rest of the design.

Perform the following steps to update the single source file:

1. Apply and save the fix to the HDL source file.
2. On the Assignments menu, click **Design Partitions Window**.
3. For the partitions that should be preserved, change the **Netlist Type** to **Post-Fit**. You can set the **Fitter Preservation Level** to either **Placement** or **Placement and Routing**. For the partition that contains the fix, you can change the netlist type to **Source File**. Making the Source File setting is optional because the Quartus II software recompiles partitions if changes are detected in a source file.
4. Click **Start Compilation** to incrementally compile the fixed HDL code. This compilation should take much less time than the initial full compilation.
5. Run simulation again to ensure that the bug is fixed, and use the Timing Analyzer report to ensure that timing results have not degraded.

Design Flow 2—Optimizing the Placement for One of Multiple Partitions in a Top-Down Compilation Flow

Use this flow when you want to optimize the results of one partition when the other partitions in the design already meet their requirements.

Example background: You have just performed a lengthy full compilation of a design that consists of multiple partitions. The Timing Analyzer reports that the clock timing requirement is not met. After some analysis, you believe that timing closure can be achieved if placement can be improved for one particular partition. You have at least three optimization techniques in mind: raising the Placement Effort Multiplier, enabling Physical Synthesis, and running the Design Space Explorer. Because these techniques all involve significant compilation time, it makes sense to apply them (or just one of them) to only the partition in question.

Perform the following steps to raise the Placement Effort Multiplier or enable Physical Synthesis:

1. On the Assignments menu, click **Design Partitions Window**.
2. For the partition in question, set the **Netlist Type** to **Post-Synthesis**. This causes the partition to be placed and routed with the new Fitter settings (but not resynthesized) during the next compilation.

3. For the remaining partitions (including the top-level entity), set the **Netlist Type** to **Post-Fit**. Set the **Fitter Preservation Level** to **Placement** to allow for the most flexibility during routing. To reduce compilation time further, use the **Placement and Routing** setting. These partitions are preserved during the next compilation.

4. Apply the desired optimization settings.

5. Click **Start Compilation** to incrementally compile the design with the new settings. During this compilation, the Partition Merge stage automatically merges the post-synthesis netlist of the critical partition with the post-fit netlists of the remaining partitions. This "merged" netlist is fed to the Fitter. The Fitter then refits only one partition. Since the effort is reduced as compared to the initial full compilation, the compilation time is also reduced.

To use Design Space Explorer, perform the following steps:

1. Repeat steps 1–3 of the previous set of steps.

2. Save the project and run Design Space Explorer.

Design Flow 3—Preserving One Critical Partition in a Multiple-Partition Design in a Top-Down Compilation Flow

Use this flow to optimize one partition by itself, and then lock the placement to preserve its results while you complete the rest of your design. For example, you can incorporate some IP that comes with instructions to perform optimization before you incorporate the rest of your custom logic.

Example background: Prior to any compilation, you have some insight into which partition will be the most critical (in terms of timing) after placement and routing. To help achieve timing closure, you decide to use the following compilation flow.

The critical partition is placed and routed by itself, with all optimizations turned on (manually or through Design Space Explorer). After timing closure is achieved for this partition, its content and placement are preserved and the remaining partitions are fit with normal or reduced optimization levels so that the compilation time can be reduced.



This flow generally works only if the critical path is contained inside the partition in question. This is one reason why both the inputs and outputs of each partition should be registered.

To implement this design flow, perform the following steps:

1. Partition the design and create floorplan location assignments.
2. For the partition expected to be critical, on the Assignments menu, click **Design Partitions Window** and set **Netlist Type** to **Source File**.
3. For the remaining partitions (other than any direct or indirect parents of the critical one), set the **Netlist Type** to **Empty**.
4. Click **Start Compilation** to compile with the desired optimizations turned on, or use Design Space Explorer.
5. Check Timing Analyzer reports to ensure that timing requirements are met. If so, proceed to step 6. Otherwise, repeat steps 4 and 5 until the requirements are met.
6. In the **Design Partitions Window**, set the **Netlist Type** to **Post-Fit** for the critical partition. Set the **Fitter Preservation Level** to **Placement and Routing** to preserve the results.
7. Change the **Netlist Type** from **Empty** to **Source File** for the remaining partitions.
8. Turn off the optimizations set in step 4, and compile the design. Turning off the optimizations at this point does not affect the fitted partition, because its Netlist Type is set to **Post-Fit**.
9. Check Timing Analyzer reports to ensure that timing requirements are met. If not, make design or option changes and repeat step 8 and step 9 until the requirements are met.



This flow is similar to a bottom-up design flow in which a module is implemented separately and is merged into the rest of the design afterwards. Refer to "Empty Partitions" on page 2–26 for more information about potential issues. Ensure that if there are any partitions representing a design file that is missing from the project, you create a placeholder wrapper file that defines the port interface.

Design Flow 4—Placing All but One Critical Partition in a Multiple-Partition Design in a Top-Down Compilation Flow

Use this flow if you want to compile your design without one timing-critical partition or a partition that requires a long compilation time, and then preserve the rest of your design when you add the last design block.

Example background: Prior to any compilation, you have some insight into which partition will be the most critical (in terms of timing) after placement and routing. To help achieve timing closure, you decide to use the following compilation flow.

Only the non-critical partitions are placed and routed initially, using floorplan location assignments. These non-critical partitions are then preserved when the critical partition is introduced into the Fitter, with various optimizations turned on (manually or through Design Space Explorer).

To implement this design flow, perform the following steps:

1. Perform partitioning and floorplan creation.
2. For the partition expected to be critical, on the Assignments menu, click **Design Partitions Window** and set the **Netlist Type** to **Empty**.
3. For the remaining partitions, set the **Netlist Type** to **Source File**.
4. Click **Start Compilation** to compile the non-critical partitions.
5. Check the Timing Analyzer report to ensure that the timing requirements are met. If so, proceed to step 6. Otherwise, make design or option changes and repeat steps 4 and 5 until the requirements are met.
6. In the **Design Partitions Window**, set the **Netlist Type** to **Post-Fit** for the processed partitions. Set the **Fitter Preservation Level** to **Placement** to allow for the most flexibility during routing.
7. Change the **Netlist Type** from **Empty** to **Source File** for the partition expected to be critical.
8. Click **Start Compilation** to compile the design with optimizations turned on, or use Design Space Explorer.
9. Check the Timing Analyzer report to ensure that the timing requirements are met. If not, make design or option changes and repeat steps 8 and 9 until the requirements are met.



This flow is similar to a bottom-up design flow, in which a module is implemented separately and merged into the rest of the design afterwards. Refer to "Empty Partitions" on page 2–26 for more information about potential issues. If there are any partitions representing a design file that is missing from the project, ensure that you create a placeholder wrapper file that defines the port interface.

Bottom-Up Incremental Design Flows

This section contains the following three bottom-up design flow examples to illustrate team-based design methodologies and IP reuse:

- "Design Flow 5—Implementing a Team-Based Bottom-Up Design Flow" on page 2–67
- "Design Flow 6—Performing Design Iteration in a Bottom-Up Design Flow" on page 2–71
- "Design Flow 7—Creating Hard-Wired Macros for IP Reuse" on page 2–73

Design Flow 5—Implementing a Team-Based Bottom-Up Design Flow

This example describes how to use incremental compilation in a bottom-up design flow.

Example background: A project consists of several lower-level subdesigns that are implemented separately by different designers. The top-level project instantiates each of these subdesigns exactly once. The subdesign designers want to optimize their designs independently and pass on the results to the project lead.

As the project lead in this scenario, perform the following steps to prepare the design for a successful bottom-up design methodology:

1. Create a new Quartus II project that will ultimately contain the full implementation of the entire design.

2. To prepare for the bottom-up methodology, create a "skeleton" of the design that defines the hierarchy for the subdesigns that will be implemented by separate designers. The top-level design implements the top-level entity in the design and instantiates wrapper files that represent each subdesign by defining only the port interfaces but not the implementation.

3. Make project-wide settings. Select the device, make global assignments for clocks and device I/O ports, and make any global signal constraints to specify which signals can use global routing resources.

4. Ensure that **Full incremental compilation** is turned on.

5. Make design partition assignments for each subdesign and set the Netlist Type for each design partition that will be imported to **Empty** in the Design Partitions window.

6. Create LogicLock regions for each of the lower-level partitions to create a design floorplan. This floorplan should consider the connectivity between partitions and estimates of the size of each partition based on any initial implementation numbers and knowledge of the design specifications.

7. On the Project menu, click **Generate Bottom-Up Design Partition Scripts**, or launch the script generator from Tcl or the command prompt.

8. Make any changes to the default script options as desired. Altera recommends that you pass all the default constraints, including LogicLock region, for all partitions and virtual pin location assignments. Altera further recommends that you add a maximum delay timing constraint for the virtual I/O connections in each partition to help timing closure during integration at the top level. If lower-level projects have not already been created by the other designers, use the partition script to set up the projects so that you can easily take advantage of makefiles.

9. Provide each lower-level designer with the Tcl file to create their project with the appropriate constraints. If you are using makefiles, provide the makefile for each partition.

As the designer of a lower-level subdesign in this example, perform the appropriate set of steps to successfully export your design, whether your design team is using makefiles, or exporting and importing the design manually.

If you are using makefiles, perform the following steps:

1. Use the **make** command and the makefile provided by the project lead to create a Quartus II project with all design constraints, and compile the project.

2. The information about which source file should be associated with which partition is not available to the software automatically, so you must specify this information in the makefile. You must specify the dependencies before the software will rebuild the project after the initial call to the makefile.

3. When you have achieved the desired compilation results and the design is ready to be imported into the top-level design, the project lead can use the **master_makefile** command to export this lower-level partition and create a Quartus II Exported Partition file, and then import it into the top-level design.

If you are not using makefiles, perform the following steps:

1. Create a new Quartus II project for the subdesign.

2. Make LogicLock region assignments and global assignments (including clock settings) as specified by the project lead.

3. Make Virtual Pin assignments for ports which represent connections to core logic instead of external device pins in the top-level module.

4. Make floorplan location assignments to the Virtual Pins so that they are placed in their corresponding regions as determined by the top-level module. This provides the Fitter with more information about the timing constraints between modules. Alternatively, you can apply timing I/O constraints to the paths that connect to virtual pins.

5. Ensure that **Full incremental compilation** is turned on and proceed to compile and optimize the design as needed.

6. When you have achieved the desired compilation results, on the Project menu, click **Export Design Partition**. The **Export Design Partition** dialog box appears.

7. Under **Netlist to export**, select the netlist type **Post-fit netlist** to preserve the placement and performance of the subdesign, and turn on **Export routing** to include the routing information if required. You can export **Post-synthesis netlist** instead if placement or performance preservation is not required.

8. Provide the Quartus II Exported Partition file to the project lead.

Finally, as the project lead in this example, perform the appropriate set of steps to import the files sent in by the designers of each lower-level subdesign partition.

If you are using makefiles, perform the following steps:

1. Use the **master_makefile** command to export each lower-level partition and create Quartus II Exported Partition files, and then import them into the top-level design.

2. The software does not have all the information about which source files should be associated with which partition, so you must specify this information in the makefile. The software cannot rebuild the project if source files change unless you specify the dependencies.

If you are not using makefiles, perform the following steps:

1. After you obtain the Quartus II Exported Partition file for each subdesign from the other designers on the team, on the Project menu, click **Import Design Partition** and specify the partition in the top-level project that is represented by the subdesign Quartus II Exported Partition file.

2. Repeat the import process described in step 1 for each partition in the design. After you have imported each partition once, select all the design partitions and use the **Reimport using latest import files at previous locations** option to import all of the files from their previous locations at one time.

Resolving Assignment Conflicts During Import

When importing the subdesigns, the project lead may become aware of some assignment conflicts. This can occur, for example, if the subdesign designers changed their LogicLock regions to account for additional logic or placement constraints, or if the designers applied I/O port timing constraints that differ from constraints added to the top-level project by the project lead. To address these conflicts, the project lead may want to take one or both of the following actions:

- Allow new assignments to be imported

- Allow existing assignments to be replaced or updated

When LogicLock region assignment conflicts occur, the project lead may take one of the following actions:

- Allow the imported region to replace the existing region
- Allow the imported region to update the existing region
- Skip assignment import for regions with conflicts

The project lead can address all of these situations using the **Advanced Import Settings** as described in "Importing Assignments and Advanced Import Settings" on page 2–37.

If the placement of different subdesigns conflict, the project lead can also set the partition's Fitter **Preservation Level to Netlist Only**, which allows the software to re-perform placement and routing with the imported netlist.

Importing a Partition to be Instantiated Multiple Times
In this variation of the scenario, one of the subdesigns is instantiated more than once in the top-level design. The designer of the subdesign may want to compile and optimize the entity once under a lower-level project, and then import the results as multiple partitions in the top-level project.

In this case, placement conflict resolution as described in "Resolving Assignment Conflicts During Import" is mandatory because the top-level partitions share the same imported post-fit netlist. If you import multiple instances of a subdesign in the top-level design, the imported LogicLock regions are automatically set to Floating status.

If you choose to resolve conflicts manually, you can use the import options and manual LogicLock assignments to specify the placement of each instance in the top-level design.

Design Flow 6—Performing Design Iteration in a Bottom-Up Design Flow

Use this flow if you want to re-optimize lower-level partitions in a bottom-up compilation by incorporating additional constraints from the integrated top-level design.

Example background: A project consists of several lower-level subdesigns that have been exported from separate Quartus II projects and imported into the top-level design in a bottom-up compilation flow. In this example, integration at the top level has failed because the timing requirements are not met. The timing requirements are met in each individual lower-level project, but critical inter-partition paths in the top level are causing timing requirements to fail.

After trying various optimizations at the top level, the project lead determines that they cannot meet the timing requirements given the current lower-level partition placements that were imported. The project lead decides to pass additional constraints to the lower-level projects to improve the placement.

To implement this design flow, perform the following steps:

1. In the top-level design, on the Project menu, click **Generate Bottom-Up Design Partition Scripts**, or launch the script generator from Tcl or the command line.

2. Because lower-level projects have already been created for each partition, turn off **Create lower-level project if one does not exist**.

3. Make any additional changes to the default script options as desired. Altera recommends that you pass all the default constraints, including LogicLock regions, for all partitions and virtual pin location assignments. Altera also recommends that you add a maximum delay timing constraint for the virtual I/O connections in each partition.

4. The Quartus II software generates Tcl scripts for all partitions, but in this scenario, you would focus on the partitions that make up the cross-partition critical paths. The following assignments are important in the script:

   - Virtual pin assignments for module pins not connected to device I/O ports in the top-level design.
   - Location constraints for the virtual pins that reflect the initial top-level placement of the pin's source or destination. These help make the lower-level placement "aware" of its surroundings in the top-level, leading to a greater chance of timing closure during integration at the top-level.
   - `INPUT_MAX_DELAY` and `OUTPUT_MAX_DELAY` timing constraints on the paths to and from the I/O pins of the partition. These constrain the pins to optimize the timing paths to and from the pins.

5. The low-level designers source the file provided by the project lead.

   - To source the Tcl script from the Quartus II GUI, on the Tools menu, click **Utility Windows** and open the Tcl console. Navigate to the script's directory, and type the following command:

     `source <`*filename*`>` ↵

- To source the Tcl script at the system command prompt, type the following command:

  ```
  quartus_cdb -t <filename>.tcl ↵
  ```

6. The lower-level designers recompile their designs with the new assignments.

7. The lower-level designers re-export their results.

8. The top-level designer re-imports the results.

9. You can now analyze the design to determine if the timing requirements have been achieved. Since the lower-level partitions were compiled with more information about connectivity at the top level, it is more likely that the inter-partition paths have improved placement which helps to meet the timing requirements.

Design Flow 7—Creating Hard-Wired Macros for IP Reuse

Use this design flow to create a hard-wired macro or IP block that can be instantiated in a top-level design. This flow provides the ability to export a design block with placement (and optionally routing) information and to import any number of copies of this pre-placed macro into another design.

Example background: An IP provider wants to produce and sell an IP core for a component to be used in higher-level systems. The IP provider wants to optimize the placement of their block for maximum performance in a specific Altera device and then pass on the placement information to their end customer. To preserve their IP, they also prefer to send a compiled netlist instead of providing the HDL source code to their customer.

The customer first specifies what Altera device they are using for this project and provides the design specifications.

As the IP provider in this example, perform the following steps to export a preplaced IP core (or hard macro):

1. Create an HDL black box wrapper file that defines the port interface for the IP core and provide the file to the customer to instantiate as an empty partition in their top-level design.

2. Create a Quartus II project for the IP core.

3. Create a LogicLock region for the design hierarchy to be exported.



Creating a floorplan using LogicLock regions is recommended although not required for the generation and use of QXP files. Using a LogicLock region for the IP core allows the customer to create an empty placeholder region to reserve space for the IP in their design floorplan. This ensures there are no conflicts with the top-level design logic, and that the IP core will not affect the timing performance of other logic in the top-level design. LogicLock regions can be effective to reduce resource utilization conflicts and to enable performance preservation. In addition, without LogicLock regions, placement can be preserved only in an absolute manner. With LogicLock regions, you can preserve placement absolutely or relative to the origin of the associated regions. This is important when a QXP file is imported for multiple partition hierarchies in the same project, because in this case the location of at least one instance in the top-level project does not match the location used by the IP provider.

4. If required, add any logic (such as PLLs or other logic that will be defined in the customer's top-level design) around the design hierarchy to be exported. If you do so, create a design partition for the design hierarchy that is to be exported as an IP core.

   For more information, refer to "Exporting a Lower-Level Block within a Project" on page 2–35.

5. Optimize the design and close timing to meet the design specifications.

6. Export the appropriate level of hierarchy into a single QXP file. Following a successful compilation of the project, you can generate a QXP file from the GUI, the command-line, or with Tcl commands:

   - If you are using the Quartus II GUI, use the **Export Design Partition** command.
   - If you are using command-line executables, run **quartus_cdb** with the `--incremental_compilation_export` option.
   - If you are using Tcl commands, run the following command: `execute_flow -incremental_compilation_export`.

7. Provide the QXP file to the customer. Note that you do not have to send any of your design source code to the customer; the design netlist as well as placement and routing information is contained within this single file.

As the customer in this example, incorporate the IP core in your design by performing the following steps:

1. Create a Quartus II project for the top-level design and instantiate a copy or multiple copies of the IP core. Add the black box wrapper file from the IP provider to your project to specify the entity name and the port interface.

2. On the Processing menu, point to Start and click **Perform Analysis & Elaboration** to identify the design hierarchy.

3. Create a design partition for each instance of the IP core (refer to "Creating Design Partitions" on page 2–100) with the Netlist Type set to Empty (refer to "Setting the Netlist Type for Design Partitions" on page 2–22).

4. You can now continue work on your part of the design and accept the IP core from the IP provider whenever it is ready.

5. Import the QXP file from the IP provider for the appropriate partition hierarchy. You can import a QXP file from the GUI, the command-line, or with Tcl commands.

   - If you are using the Quartus II GUI, use the **Import Design Partition** command.
   - From the command-line, run **quartus_cdb** with the `--incremental_compilation_import` option.
   - With Tcl commands, run the following command: `execute_flow -incremental_compilation_import`.

6. You can set the imported LogicLock regions to floating or move them to a new location, with the relative locations of the region contents preserved. Routing information is preserved whenever possible.

   The Fitter ignores relative placement assignments if the LogicLock region's location in the top-level design is not compatible with the locations exported in the QXP file.

7. You can control whether to preserve the imported netlist only, placement, or placement and routing (if the placement or placement and routing information was exported in the QXP file) with the Fitter Preservation Level.

   By default, the software preserves the absolute placement and routing of all nodes in the imported netlist if you choose to preserve placement and routing. However, if you use the same QXP files for multiple partitions in the same project, the software preserves the relative placement for each of the imported modules (relative to the origin of the LogicLock region).



If the IP provider did not define a LogicLock region in the exported partition, the software preserves absolute placement locations and this leads to placement conflicts if the partition is imported for more than one instance.

Incremental Compilation Restrictions

This section documents the restrictions and limitations that you may encounter when using incremental compilation, including interactions with other Quartus II features. Some restrictions apply to both top-down and bottom-up design flows, while some additional restrictions apply only to bottom-up design flows.

The following restrictions and limitations are covered:

- "Using Incremental Compilation with Quartus II Archive Files" on page 2–77
- "OpenCore Plus MegaCore Functions in Bottom-Up Flows" on page 2–78
- "SignalProbe Pins and Engineering Change Management with the Chip Planner" on page 2–78
- "SignalTap II Embedded Logic Analyzer in Bottom-Up Compilation Flows" on page 2–80
- "HardCopy Compilation Flows" on page 2–82
- "Restrictions on Megafunction Partitions" on page 2–84
- "Routing Preservation in Bottom-Up Compilation Flows" on page 2–84
- "Bottom-Up Design Partition Script Limitations" on page 2–84
- "Register Packing and Partition Boundaries" on page 2–87
- "I/O Register Packing" on page 2–87

Using Incremental Synthesis Only Instead of Full Incremental Compilation

You can turn on incremental compilation for only the synthesis stage of compilation to perform incremental synthesis, with no incremental place-and-route. This mode is not recommended for new projects because it is not compatible with certain Quartus II design flows, such as formal verification and incremental SignalTap II verification.

To use incremental synthesis only, you can follow the steps for full incremental compilation, but turn on the **Incremental synthesis only (Can reduce compilation time for a design with partition assignments)** option on the **Incremental Compilation** page under **Compilation Process Settings** in the **Settings** dialog box.

In this mode, the Fitter uses a flattened netlist without partition boundaries, so the design is always replaced and rerouted. The difference between this flow and the one shown in Figure 2–2 on page 2–7 is that the partition merge stage does not accept post-fit netlists produced by the Fitter, and the Fitter does not compile partitions separately. The following differences exist in the impact of incremental synthesis only as compared to full incremental compilation:

- Compilation time reduction is limited to Quartus II integrated synthesis.
- You cannot preserve placement and routing, therefore the feature does not preserve partition timing performance.
- A partition is automatically resynthesized whenever you make a change to the source code or any synthesis assignments (changes to synthesis or fitting assignments do not trigger an automatic recompilation with Full Incremental Compilation).

Preserving Exact Timing Performance

Timing performance might change slightly in the top-level design when all partitions are incorporated due to differences between the separate partitions and the full design. For example, there may be parasitic effects or crosstalk that was not present in the initial compilation with only part of the design. Additional fan-out on routing lines can also degrade timing performance. To ensure that the design will meet performance when all partitions are present, only approximately 2% margin is required. This applies to both bottom-up and top-down methodologies. The Fitter automatically works to achieve more than 2% margin when compiling any design.

Using Incremental Compilation with Quartus II Archive Files

The post-synthesis and post-fitting netlist information for each design partition is stored in the project database. When you archive a project, the database information is not included in the archive unless you include the database files in the Quartus II Archive file (**.qar**). In addition, when you import a design partition into a top-level design, the lower-level design netlist is stored in the project database for the top-level design (the top-level project does not use the original source files or the Quartus II Exported Partition file). If you archive the top-level project, the imported design information is not included unless the database files are included in the Quartus II Archive file.

Altera recommends that you turn on **Include database from compilation and simulation** in the **Archive Project** dialog box if any form of incremental compilation is used so that compilation results are preserved.

Formal Verification Support

You cannot use design partitions if you are creating a netlist for a formal verification tool.

OpenCore Plus MegaCore Functions in Bottom-Up Flows

You can use OpenCore Plus MegaCore® functions in top-down incremental compilation flows beginning with the Quartus II software version 7.1. You cannot export partitions containing OpenCore Plus MegaCore functions, so you cannot use OpenCore Plus functions in a bottom-up design flow. Include any OpenCore IP functions in your top-level Quartus II project.

Importing Encrypted IP Cores in Bottom-Up Flows

Proper license information is required to compile encrypted IP cores. The license assignment is imported in to the top-level project when a design is imported as a QXP file. However, the license assignment contains an absolute path to the licensed IP source files. Therefore, the QXP file usually works correctly only if imported into a top-level project on the same computer as the lower-level project.

To avoid this problem, you can include this partition in the top-level project instead of importing it, because IP cores generally do not require additional changes by a designer in the project team. You can set the partition that contains the core to Post-Fit after the first compilation to reduce future compilation times, because the partition will not be changing in any subsequent compilation. You can also set the partition to Empty to exclude the IP core from the database until you are ready to compile the entire design.

If you do want to import an encrypted IP core, copy the encrypted IP source files to the top-level project's computer in exactly the same path structure. For example, if the IP encrypted source file was **d:/work/my_encrypted_file.vhd**, the top-level designer that imports the QXP file must create the same folder and place the file in this location.

SignalProbe Pins and Engineering Change Management with the Chip Planner

When you create SignalProbe pins or use the Resource Property Editor to make changes due to engineering change orders (ECOs) after performing a full compilation, recompiling the entire design is not necessary. These changes are made directly to the netlist without performing a new placement and routing. You can preserve these changes using a post-fit

netlist with placement and routing. When a partition is recompiled, SignalProbe pins and ECO changes in unaffected partitions are preserved.



For more information about using the SignalProbe feature to debug your design, refer to the *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*. For more information about using the Chip Planner and the Resource Property Editor to make ECOs, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

To preserve SignalProbe pins or ECO changes, the partitions must be set to a Netlist Type of Post-fit with the Fitter Preservation Level set to Placement and routing. If any partitions with SignalProbe pins or ECO changes are set to post-fit without routing or to netlist only, the software issues a warning and internally uses the post-fit netlist with placement and routing. If the partitions are set to use the source code or a post-synthesis netlist, the software issues a warning and the post-fit SignalProbe pins or ECO changes are not included in the new compilation. However, partitions can become linked due to the SignalProbe pins or ECO changes, as described below, in which case all linked partitions inherit the netlist type from the linked partition with the highest level of preservation.

Linked Partitions Due to SignalProbe Pins or ECO Changes

If ECO changes affect more than one partition or the connection between any partitions, the partitions become linked. All of the higher-level "parent" partitions up to their nearest common parent are also linked. In this case, the connection between the partitions is actually defined outside of the two partitions immediately affected, so all the partitions must be compiled together. All linked partitions use the same netlist type, and they inherit the netlist type from the linked partition with the highest level of preservation.

When a SignalProbe pin is created, it affects the partition that contains the node being probed. In addition, any pipeline registers are created in the same partition as the node being probed. The SignalProbe output pin is assigned to the top-level partition. Therefore, there is a new connection formed between the top-level partition and the lower-level partition that is being probed. Because of this connection, the lower-level partition being probed and all of the higher-level "parent" partitions up to the top level become linked. All linked partitions use the same netlist type, and they inherit the netlist type from the linked partition with the highest level of preservation.

When partitions are linked, they can change which netlists are preserved when you recompile the design, as follows:

- If all the linked partitions are set to use the source code or a post-synthesis netlist, the partitions are refit as normal. In this case, the SignalProbe pins or ECO changes are not included in the new netlists, so you must reapply the changes in the Change Manager.
- If any of the linked partitions is set to the Post-Fit netlist type, and there are no source code changes, the software issues a warning and internally uses the post-fit netlist with placement and routing for all linked partitions. By preserving the appropriate post-fit netlists, the software can preserve the SignalProbe pins or ECO changes.
- If any of the linked partitions is set to the Post-Fit (Strict) netlist type, the software issues a warning and internally uses the post-fit netlist with placement and routing for all linked partitions, regardless of any source code changes. By preserving the appropriate post-fit netlists, the software can preserve the SignalProbe pins or ECO changes. Note that in this case, source code changes in any of the linked partitions are not included in the new netlist.
- If any of the linked partitions is recompiled due to a change in source code, the software issues a warning and recompiles the other linked partition(s) as well. When this occurs, the SignalProbe pins or ECO changes are not included in the new netlist, so you must reapply the changes in the Change Manager.

Exported Partitions

In a bottom-up incremental compilation, the exported netlist includes all currently saved SignalProbe pins and ECO changes. This might require flattening and combining lower-level partitions in the child project to avoid partition boundary violations at the top level. After importing this netlist, changes made in the lower-level partition do not appear in the Change Manager at the top level.

If you make any ECO changes that affect the interface to the lower-level partition, the software issues a warning message during the export process that this netlist will not work in the top-level design without modifying the top-level HDL code to reflect the lower-level change.

SignalTap II Embedded Logic Analyzer in Bottom-Up Compilation Flows

You can use the SignalTap® II Embedded Logic Analyzer in any project that you can compile and program into an Altera device. You cannot export a lower-level project that uses a SignalTap II File (**.stp**) for the SignalTap II Logic Analyzer in a bottom-up incremental compilation flow. You must disable the SignalTap II feature and recompile the design

before you export the design as a partition. You can instantiate the SignalTap II Megafunction directly in your lower-level design (instead of using an **.stp** file) and export the design to the top level in a bottom-up flow.

You can tap any nodes in a Quartus II project, including nodes imported from other projects. Use the appropriate filter in the Node Finder to find your node names. Use **SignalTap II: post-fitting** if the Netlist Type is Post-Fit to incrementally tap node names in the post-fit netlist database. Use **SignalTap II: pre-synthesis** if the Netlist Type is Source File to make connections to the source file (pre-synthesis) node names when you synthesize the partition from the source code.



For details about using the SignalTap II logic analyzer in an incremental design flow, refer to the *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Logic Analyzer Interface in Bottom-Up Compilation Flows

You can use the Logic Analyzer Interface in any project that you can compile and program into an Altera device. You cannot export a lower-level project that uses the Logic Analyzer Interface in a bottom-up incremental compilation flow. You must disable the Logic Analyzer Interface feature and recompile the design before you export the design as a partition.



For more information about the Logic Analyzer Interface, refer to the *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*.

Migrating Projects with Design Partitions to Different Devices

Partition assignments are still valid if you migrate to a different device density or family. LogicLock region size is valid if you migrate to a device in the same family, but the origin location is not valid. Specific floorplan assignments are not valid for different devices or families because the location coordinates change between devices.

Post-synthesis netlists are valid if you migrate to a different-sized device in the same family. Post-fit netlists are not valid if you migrate to a different device density or family.

HardCopy Compilation Flows

HardCopy APEX and HardCopy Stratix Devices

Incremental compilation with the Quartus II software is not supported for HardCopy APEX or HardCopy Stratix design flows.

HardCopy II Migration Flows

Top-down incremental compilation is supported for the base family in HardCopy II migration flows for both the Stratix II first and HardCopy II first flows. Design partitions are migrated to the companion device. LogicLock regions are suggested for design partitions but are not migrated to the companion device, due to the different device architecture. However, you can not make changes to the design after migration because the design would not match the compilation results for the base family.

The Netlist only preservation level is not supported for Post-fit netlists for Stratix II or HardCopy II device compilations when there is a migration device set (that is, for HardCopy II device compilations with a Stratix II migration device, or Stratix II device compilations with a HardCopy II migration device).

Bottom-up incremental compilation is not supported in HardCopy II or Stratix II device compilations when there is a migration device setting. The Revision Compare feature requires that the HardCopy II and FPGA netlists are the same. Therefore, all operations performed on one revision must also occur on the other revision. This is accomplished by logging all operations and replaying them on the other revision. Unfortunately, using the bottom-up flow and importing partitions does not support this requirement. You can often use a top-down flow with Empty partitions to implement behavior similar to bottom-up flows.

HardCopy II Stand-Alone Compilations

You can use both top-down and bottom-up incremental compilation for stand-alone HardCopy II compilations.

Routing preservation is not supported for HardCopy II devices. Therefore, the **Placement and Routing** preservation level is not available, and routing cannot be exported in the bottom-up flow.

Assignments Made in HDL Source Code in Bottom-Up Flows

Assignments made with I/O primitives or the `altera_attribute` HDL synthesis attribute in lower-level partitions are not currently honored at the top level in a bottom-up flow. The assignments are processed at the top level, but cannot always be applied to the netlist database after import. Fitter-related assignments (such as I/O termination setting) can be applied correctly if you use a post-synthesis QXP file.

Compilation Time with Physical Synthesis Optimizations

If Physical Synthesis is turned on, the optimizations run whenever there is any partition placement that is not fixed with a post-fit netlist. For example, when using the SignalTap II logic analyzer, there is an automatic partition created for the SignalTap II instance which does not have its placement preserved. Physical synthesis cannot make any changes to partitions that are set to post-fit; however, it does still analyze the netlist as whole. Therefore, the compilation time is not reduced as much if physical synthesis optimizations are turned on.

You can set partitions to Empty to reduce compilation time if you want to use physical synthesis for other partitions. You can go back to the Post-fit netlist type directly from Empty, so the previous fitting results can be reused when you want to include all partitions in the netlist. This method works best if you assign each Empty partition to a LogicLock region with the Reserved property, so that no other logic is placed in that region of the device floorplan when the design is recompiled.

You can also turn off physical synthesis if you are recompiling a partition which does not require physical synthesis optimizations. For example, when using the SignalTap II Logic Analyzer on a design that has all partitions using post-fit netlists, you can turn off physical synthesis to reduce compilation time. You can also compile critical partitions that require Physical Synthesis first, and close timing for those partitions. If those partitions do not require any logic changes, you can set the critical partitions to post-fit and then subsequent compilations can have physical synthesis turned off. Be sure to turn the option on again if you make design changes to timing-critical partitions and want to recompile the new logic with physical synthesis optimizations.

Restrictions on Megafunction Partitions

The Quartus II software does not support partitions for megafunction instantiations. If you use the MegaWizard Plug-In Manager to customize a megafunction variation, the MegaWizard-generated wrapper file instantiates the megafunction. You can create a partition for the MegaWizard-generated megafunction custom variation wrapper file.

The Quartus II software does not support the creation of a partition for inferred megafunctions (that is, where the software infers a megafunction to implement logic in your design). If you have a module or entity for the logic that is inferred, you can create a partition for that hierarchy level in the design.

The Quartus II software does not support creation of a partition for any Quartus II internal hierarchy that is dynamically generated during compilation to implement the contents of a megafunction.

Routing Preservation in Bottom-Up Compilation Flows

There are some cases in which routing information cannot be preserved exactly, especially in bottom-up compilation, because of legality in the device architecture. For example, when multiple partitions are imported, there may be routing conflicts because you cannot pre-assign routing for each lower-level block. In addition, if an imported LogicLock region is moved in the top-level design, the relative placement of the nodes is preserved but the routing may not be preserved.

Bottom-Up Design Partition Script Limitations

The Quartus II software has some limitations related to bottom-up design partition scripts.

Synopsys Design Constraint (SDC) Files for the TimeQuest Timing Analyzer

As described in "Importing Assignments and Advanced Import Settings" on page 2–37, timing assignments made for the TimeQuest Timing Analyzer in an SDC file are currently not imported into the top-level project. You should manually ensure that the top-level project includes all of the timing requirements for the entire project.

If you want to copy lower-level SDC files to the top-level project, consider prefixing lower-level constraints with a variable that describes the constraint's location in the design hierarchy. Then, when you copy the file to the top-level design, you can set the variable to provide the hierarchy path to the lower-level partition in the top-level design.

Wildcard Support in Bottom-Up Design Partition Scripts

When applying constraints with wildcards, wildcards are not analyzed across hierarchical boundaries. For example, an assignment could be made to these nodes: **Top|A:inst|B:inst|***, where **A** and **B** are lower-level partitions, and hierarchy **B** is a child of **A**, that is **B** is instantiated in hierarchy **A**. This assignment is applied to modules **A**, **B** and all children instances of **B**. However, the assignment **Top|A:inst|B:inst*** is applied to hierarchy **A**, but is not applied to the **B** instances because the single level of hierarchy represented by **B:inst*** is not expanded into multiple levels of hierarchy. To avoid this issue, ensure that you apply the wildcard to the hierarchical boundary if it should represent multiple levels of hierarchy.

When using the wildcard to represent a level of hierarchy, only single wildcards are supported. This means assignments such as **Top|A:inst|*|B:inst|*** are not supported. The Quartus II software issues a warning in these cases.

Derived Clocks and PLLs in Bottom-Up Design Partition Scripts

If a clock in the top level is not directly connected to a pin of a lower-level partition, then the lower-level partition does not receive assignments and constraints from the top-level pin in the design partition scripts.

This issue is of particular importance for clock pins that require timing constraints and clock group settings. Problems can occur if your design uses logic or inversion to derive a new clock from a clock input pin. Make appropriate timing assignments in your lower-level Quartus II project to ensure that clocks are not unconstrained.

In addition, if you use a PLL in your top-level design and connect it to lower-level partitions, the lower-level partitions do not have information about the multiplication or phase shift factors in the PLL. Make appropriate timing assignments in your lower-level Quartus II project to ensure that clocks are not unconstrained or constrained with the incorrect frequency. Alternately, you can manually duplicate the top-level derived clock logic or PLL in the lower-level design file to ensure that you have the correct multiplication or phase shift factors, compensation delays and other PLL parameters for complete accurate timing analysis. Create a design partition for the rest of the lower-level design logic that will be exported to the top level. When the lower-level design is complete, export just the partition that contains the relevant logic with the "Exporting a Lower-Level Block within a Project" on page 2–35 feature.

Virtual Pin Timing Assignments in Bottom-Up Design Partition Scripts

The design partition scripts use `INPUT_MAX_DELAY` and `OUTPUT_MAX_DELAY` assignments to specify the inter-partition delays associated with input and output pins which would not otherwise be visible to the project. These assignments require that the software specify the clock domain for the assignment, and the software sets this clock domain to '*'.

This clock domain assignment means that there may be some paths constrained and reported by the timing analysis engine that are not required.

To restrict which clock domains are included in these assignments, edit the generated scripts or change the assignments in your lower-level Quartus II project. In addition, because there is no known clock associated with the delay assignments, the software assumes the worst-case skew, which makes the paths seem more timing critical than they are in the top-level design. To make the paths appear less timing-critical, lower the delay values from the scripts. If required, you can also enter negative numbers for input and output delay values.

Top-Level Ports that Feed Multiple Lower-Level Pins in Bottom-Up Design Partition Scripts

When a single top-level I/O port drives multiple pins on a lower-level module, it unnecessarily restricts the quality of the synthesis and placement at the lower-level. This occurs because in the lower-level design, the software must maintain the hierarchical boundary and cannot use any information about pins being logically equivalent at the top level. In addition, because I/O constraints are passed from the top-level pin to each of the children, it is possible to have more pins in the lower level than at the top level, and these pins use the top-level I/O constraints and placement options that might make them impossible to place at the lower-level. The software avoids this situation whenever possible, but it is best to avoid this design practice to avoid these potential problems. Restructure your design so that the single I/O port feeds the design partition boundary, and then the connection is split into multiple signals within the lower-level partition.

Register Packing and Partition Boundaries

The Quartus II software automatically performs register packing during compilation. However, when incremental compilation is enabled, logic in different partitions cannot be packed together because partition boundaries prevent cross-boundary optimization. (Refer to "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–46 for more information.) This restriction applies to all types of register packing, including I/O cells, DSP blocks, sequential logic, and unrelated logic.

I/O Register Packing

Cross-partition register packing of I/O registers is allowed in certain cases where your input and output pins exist in the top hierarchy level (and the Top partition), but the corresponding I/O registers exist in other partitions.

The following specific circumstances are required for cross-partition register packing of input pins:

- The input pin feeds exactly one register
- The path between the input pin and the register includes only input ports of partitions that have one fan-out each

The following specific circumstances are required for cross-partition register packing of output registers:

- The register feeds exactly one output pin
- The output pin is fed by only one signal
- The path between the register and the output pin includes only output ports of partitions that have one fan-out each

Output pins with an output enable signal cannot be packed into the device I/O cell if the output enable logic is part of a different partition from the output register. To allow register packing for output pins with an output enable signal, structure your HDL code or design partition assignments so that the register and the tri-state logic are defined in the same partition.

Bidirectional pins are handled in the same way as output pins with an output enable. If the registers that need to be packed are in the same partition as the tri-state logic, then register packing can be performed.

The restrictions on tri-state logic are due to the fact that the I/O atom (device primitive) is created as part of the partition that contains the tri-state logic. If an I/O register and its tri-state logic are contained in the same partition, the register can always be packed with the tri-state logic into the I/O atom. The same cross-partition register packing restrictions also apply to I/O atoms for input and output pins. The I/O atom must feed the I/O pin directly with exactly one signal and the path between the I/O atom and the I/O pin must include only ports of partitions that have one fan-out each.

Examples of I/O Register Packing Across Partition Boundaries

The following examples provide detailed explanations for various I/O and partition configurations. The examples use BDF schematics to illustrate the design logic.

Example 1—Output Register in Partition Feeding Output Pin
In this example, a subdesign contains a single register, as shown in Figure 2–16. As shown in Figure 2–17, the top-level design instantiates the subdesign with a single fan-out directly feeding an output pin, and designates the subdesign as a separate design partition.

Figure 2–16. Subdesign with One Register, Designated as a Separate Partition

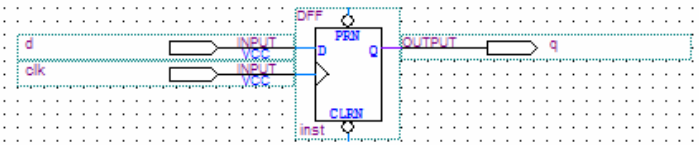
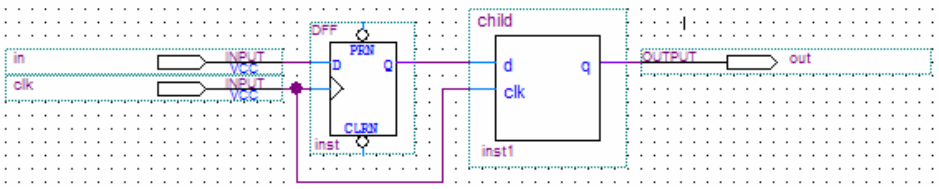


Figure 2–17. Top-level Design Instantiating the Subdesign in Figure 2–16 as an Output Register

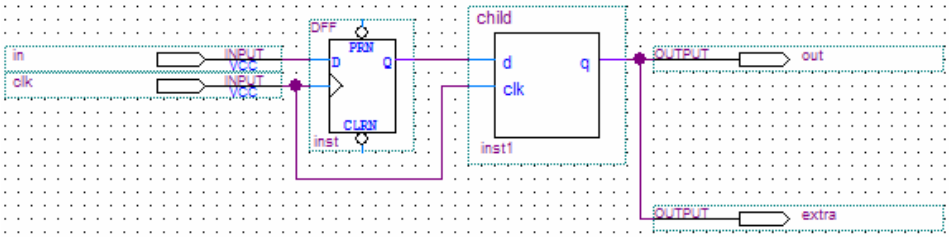


The Quartus II software performs cross-partition register packing if there is a Fast Output Register assignment on pin `out`. This type of cross-partition output register packing is permitted because the port interface of the subdesign partition does not need to be changed and the partition port feeds an output pin directly.

Example 2—Output Register in Partition Feeding Multiple Output Pins

In this example, a subdesign designated as a separate partition contains a register as in Figure 2–16. The top-level design instantiates the subdesign as an output register with more than one fan-out signal, as shown in Figure 2–18.

Figure 2–18. Top-level Design Instantiating the Subdesign in Figure 2–16 with Two Output Pins



In this case, the software does not perform output register packing. If there is a Fast Output Register assignment on pin `out`, the software issues a warning that the Fitter can't pack the node to an I/O pin because the node and the I/O cell are connected across a design partition boundary.

This kind of cross-partition register packing is not permitted because it would require modification to the interface of the subdesign partition. In order to perform incremental compilation, the interface of design partitions must be preserved.

To allow the software to pack the register in the subdesign from Figure 2–16 with the output pin `out` in Figure 2–18, make one of the following changes:

- Remove the design partition assignment to the subdesign. This allows the Fitter to perform all cross-hierarchy optimizations. However, it prevents you from using incremental compilation for this block of hierarchy. A good design partition should have a well-defined interface so that the Fitter does not have to perform cross-boundary optimizations.
- Restructure your HDL code to place the register in the same partition as the output pin. The simplest option is to move the register from the subdesign partition into the partition containing the output pin. This guarantees that the Fitter can optimize the two nodes without violating any partition boundaries.
- Restructure your HDL code so the register feeds only one output pin. Turn off the Analysis and Synthesis setting **Remove Duplicate Registers**. Duplicate the register in your subdesign HDL as in

Figure 2–19 so that each register feeds only one pin, then connect the extra output pin to the new port in the top-level design as shown in Figure 2–20. This converts the cross-partition register packing into the simplest case where the register has a single fan-out.

Figure 2–19. Modified Subdesign from Figure 2–16 with Two Output Registers and Two Output Ports

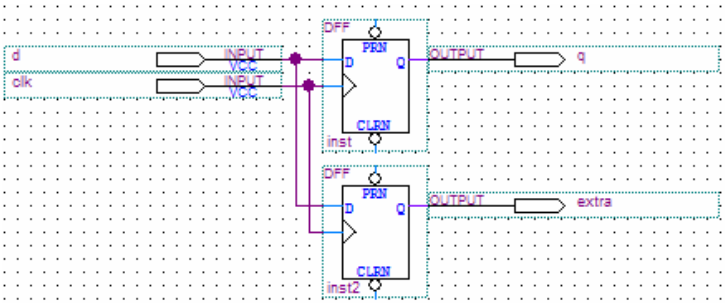
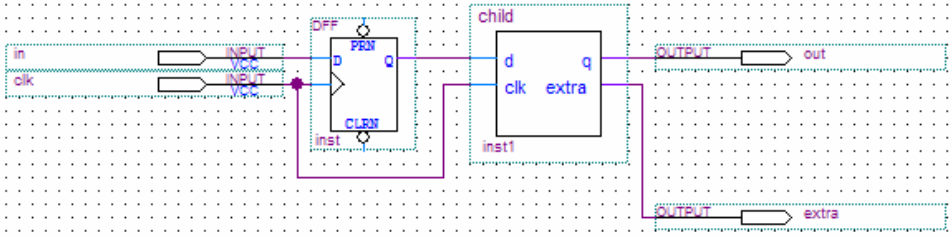


Figure 2–20. Modified Top-Level Design from Figure 2–18 Connecting Two Output Ports to Output Pins



Example 3—Output Register, Output Enable Register and Tri-State Logic in Partition Feeding Output Pin

In this example, a subdesign designated as a separate partition contains an output register, an output enable register, and the tri-state logic to drive the output pin, as shown in Figure 2–21. The top-level design instantiates the subdesign with a single fan-out directly feeding an output pin, as shown in Figure 2–22.

Figure 2–21. Subdesign with Output Register, Output Enable Register and Tri-State Logic, Designated as a Separate Partition

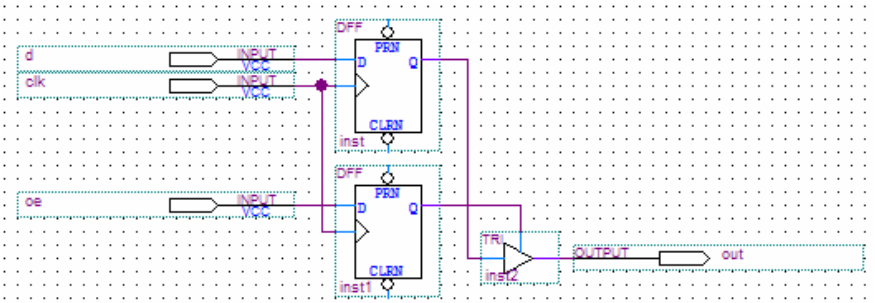
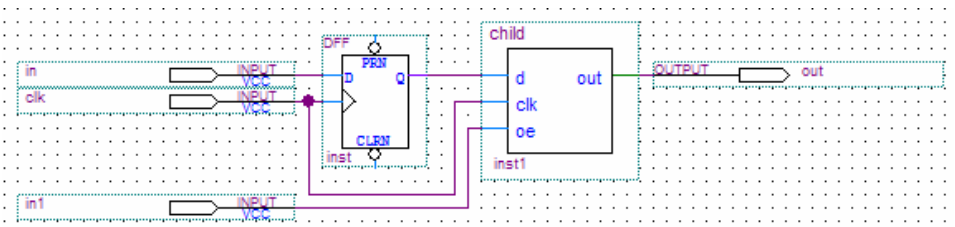


Figure 2–22. Top-level Design Instantiating the Subdesign in Figure 2–21



The Quartus II software performs cross-partition register packing if there is a Fast Output Register assignment, Fast Output Enable Register assignment, or both, on pin `out`. This kind of cross-partition output register packing is permitted because the port interface of the subdesign partition does not need to be changed, no logic needs to be optimized across the partition boundary, and the partition port feeds an output pin directly.

Example 4—Output Register, Output Enable Register, or Both, in Partition Feeding Tri-State Output Pin

In this example, a subdesign designated as a separate partition contains two registers, as shown in Figure 2–23. The top-level design instantiates the subdesign with the registers driving the output and the output enable signal for an output pin, as shown in Figure 2–24.

Figure 2–23. Subdesign with Two Registers, Designated as a Separate Partition

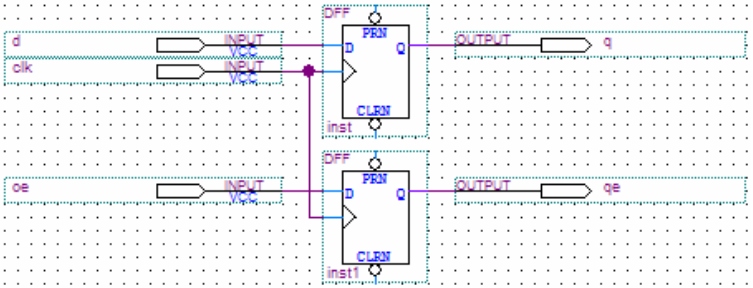
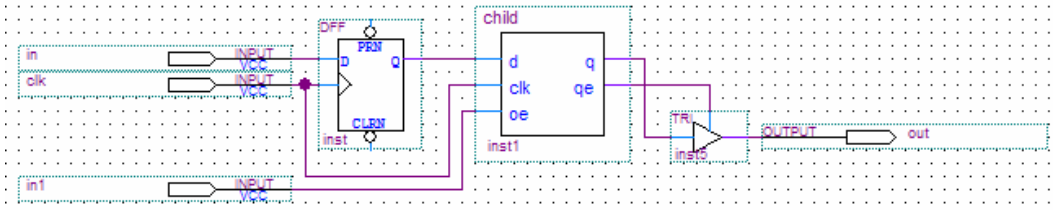


Figure 2–24. Top-level Design Instantiating the Subdesign in Figure 2–25 to Drive Output Enable Logic



In this case, the software cannot perform register packing. If there is a Fast Output Register or Fast Output Enable Register assignment on pin `out`, the software issues a warning that the Fitter cannot pack the node to an I/O pin because the node and the I/O cell are connected across a design partition boundary.

The same restrictions apply in the case that the top-level design includes either the output register or the output enable register as well as the tri-state logic. The software cannot pack the register that is part of the subdesign partition into the I/O register.

This type of register packing is not permitted because it would require moving logic across a design partition boundary to place into a single I/O device atom. To perform register packing, either the registers must be moved out of the subdesign partition or the tri-state logic must be moved into the subdesign partition. In order to guarantee correctness of the design with subsequent incremental compilations, the contents of design partitions must be preserved.

To allow the software to pack the output register, output enable register, or both, in the subdesign from Figure 2–23 with the output pin `out` in Figure 2–24, make one of the following changes:

- Remove the design partition assignment to the subdesign. This allows the Fitter to perform all cross-hierarchy optimizations. However, it prevents you from using incremental compilation for this block of hierarchy. A good design partition should have a well-defined interface so that the Fitter does not need to perform cross-boundary optimizations.
- Restructure your HDL code to place the register in the same partition as the output pin. The simplest option is to move the register from the subdesign partition into the top-level partition containing the output pin. This guarantees that the Fitter can optimize the two nodes without violating any partition boundaries.
- Restructure your HDL code so the register and the tri-state logic are contained in the same partition. Move the tri-state logic from the top-level block into the subdesign with both registers as shown in Figure 2–21. Then connect the subdesign to an output pin in the top-level design, as shown in Figure 2–22.

Example 5—Bidirectional Logic in Partition Feeding Bidirectional Pin
The behavior for bidirectional pins is similar to that of an output pin with an output enable. To allow register packing, the registers must be included in the same partition as the tri-state logic that drives the bidirectional pin.

In this example, a subdesign designated as a separate partition contains three registers and the tri-state logic for a bidirectional pin, as shown in Figure 2–25. The top-level design instantiates the subdesign with ports feeding bidirectional and output pins, as shown in Figure 2–26.

Figure 2–25. Subdesign with Three Registers and Tri-State Logic, Designated as a Separate Partition

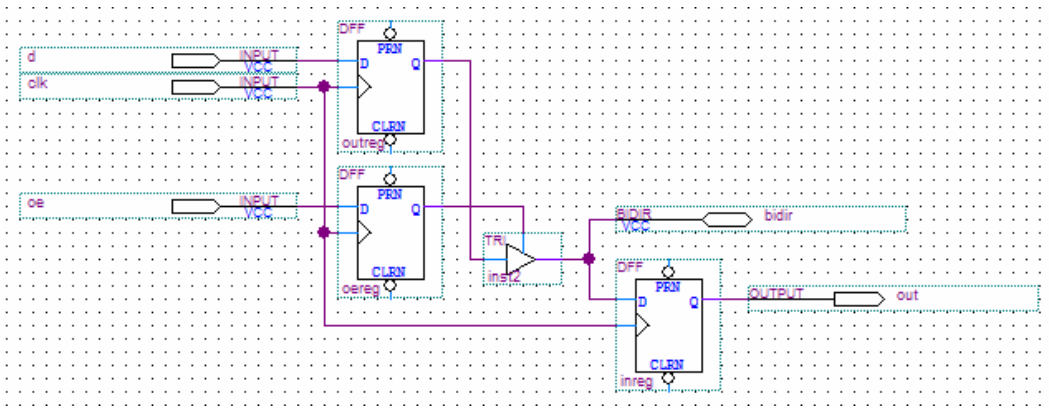
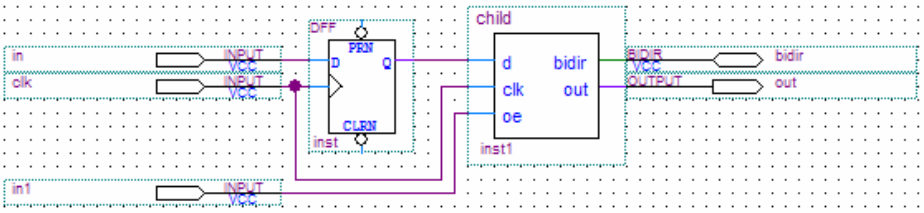


Figure 2–26. Top-level Design Instantiating the Subdesign in Figure 2–28



The Quartus II software performs cross-partition register packing if there is a Fast Output Register, Fast Output Enable Register, or Fast Input Register assignment on pin `bidir`. This type of cross-partition output register packing is permitted because the port interface of the subdesign partition does not need to be changed and the partition port feeds a bidirectional pin directly.

Registers cannot be packed in designs that have the registers and tri-state logic in different partitions. The situations described in "Example 4—Output Register, Output Enable Register, or Both, in Partition Feeding Tri-State Output Pin" on page 2–91 apply similarly to bidirectional pins if you replace the output pin `out` with a bidirectional pin in the top-level design.

Example 6—Input Register in Partition Fed by Input Pin
In this example, a subdesign contains a single register, as shown in Figure 2–27. The top-level design instantiates the subdesign with a single `fanin` directly fed by an input pin, as shown in Figure 2–28, and designates the subdesign to be a separate design partition.

Figure 2–27. Subdesign with One Register, Designated as a Separate Partition

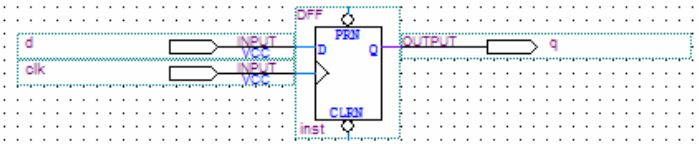
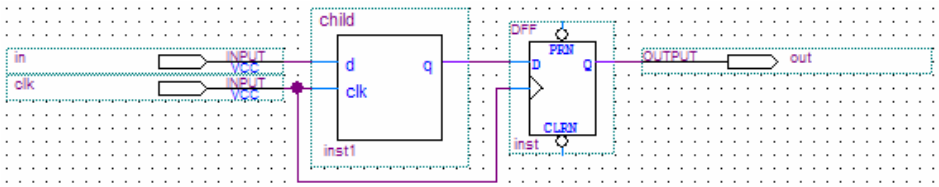


Figure 2–28. Top-level Design Instantiating the Subdesign in Figure 2–27 as an Input Register

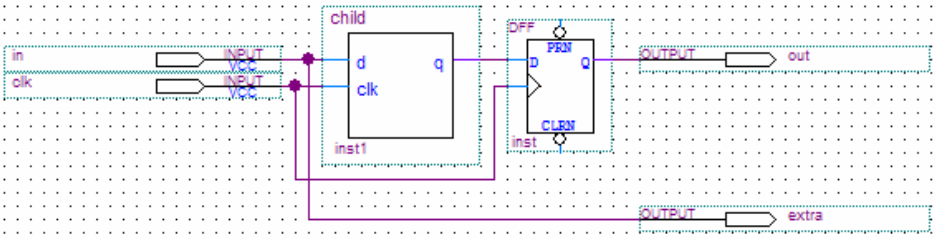


The Quartus II software performs cross-partition register packing if there is a Fast Input Register assignment on pin `in`. This type of cross-partition output register packing is permitted because the port interface of the subdesign partition does not have to be changed and the partition port is fed by an input pin directly.

Example 7—Input Register in Partition Fed by Input with Multiple Fan-Out

In this example, a subdesign designated as a separate partition contains a register as in Figure 2–27. The top-level design instantiates the subdesign as an input register but the input pin also feeds another destination, as shown in Figure 2–29.

Figure 2–29. Top-level Design Instantiating the Subdesign in Figure 2–27 as an Input Register for a Pin with Two Destinations



In this case, the software does not perform input register packing. If there is a Fast Input Register assignment on pin `in`, the software issues a warning that the Fitter cannot pack the node to an I/O pin because the node and the I/O cell are connected across a design partition boundary.

This type of cross-partition register packing is not permitted because it would require modification to the interface of the subdesign partition. In order to perform incremental compilation, the interface of design partitions must be preserved.

To allow the software to pack the register in the subdesign from Figure 2–27 with the input pin `in` in Figure 2–29, make one of the following changes:

- Remove the design partition assignment to the subdesign. This allows the Fitter to perform all cross-hierarchy optimizations. However, it also prevents you from using incremental compilation for this block of hierarchy. A good design partition should have a well-defined interface so that the Fitter does not have to perform cross-boundary optimizations.
- Restructure your HDL code to place the register in the same partition as the input pin. The simplest option is to move the register from the subdesign partition into the partition containing the input pin. This guarantees that the Fitter can optimize the two nodes without violating any partition boundaries.

Example 8—Inverted Input Register in Partition Fed by Input Pin
In this example, a subdesign designated as a separate partition contains an inverted register as in Figure 2–30. The top-level design instantiates the subdesign as an input register, as shown in Figure 2–31.

Figure 2–30. Subdesign with an Inverted Register, Designated as a Separate Partition

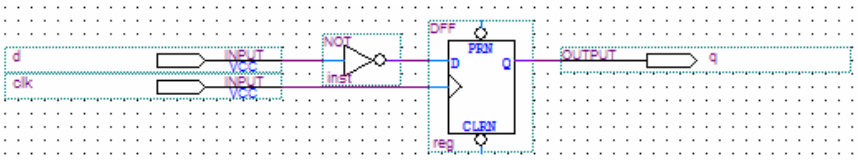
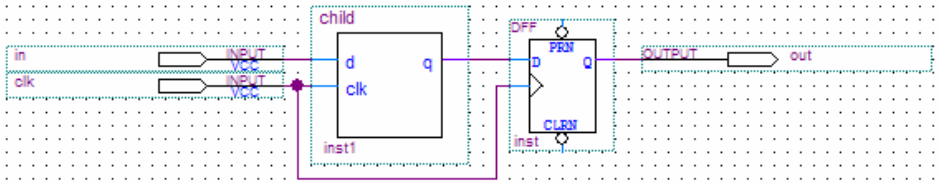


Figure 2–31. Top-level Design Instantiating the Subdesign in Figure 2–30 as an Input Register



The Quartus II software performs cross-partition register packing if there is a Fast Input Register assignment on pin `in`. This kind of cross-partition input register packing is permitted because the software can implement the logic for the inversion with the input register inside the partition, and then the partition port is fed by an input pin directly.

Example 9—Input Register in Partition Fed by Inverted Input Pin, or Output Register in Partition Feeding Inverted Output Pin

In this example, a subdesign designated as a separate partition contains a register as in Figure 2–32. The top-level design in Figure 2–33 instantiates the subdesign as an input register with the input pin inverted. The top-level design in Figure 2–34 instantiates the subdesign as an output register with the signal inverted before feeding an output pin.

Figure 2–32. Subdesign with One Register, Designated as a Separate Partition

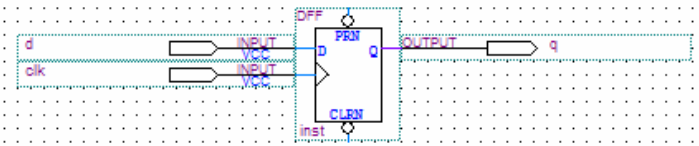


Figure 2–33. Top-level Design Instantiating the Subdesign in Figure 2–32 as an Input Register with an Inverted Input Pin

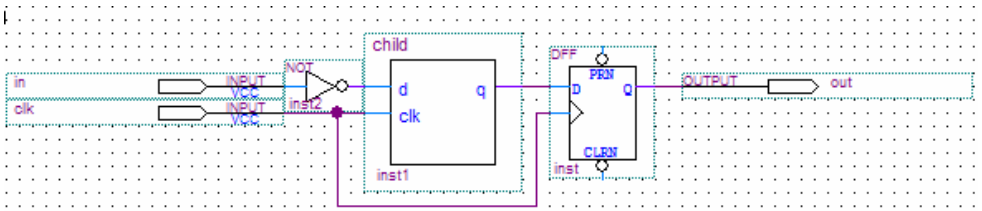
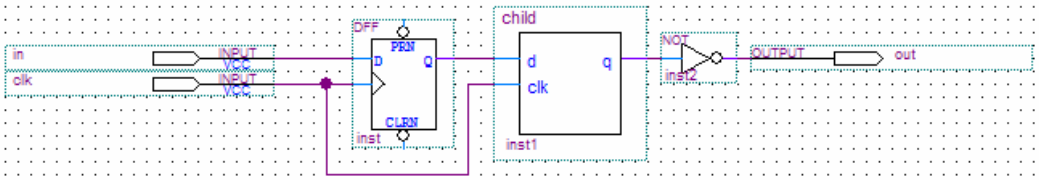


Figure 2–34. Top-level Design Instantiating the Subdesign in Figure 2–33 as an Output Register Feeding an Inverted Output Pin



In these cases, the software does not perform register packing. If there is a Fast Input Register assignment on pin `in` in Figure 2–33 or a Fast Output Register assignment on pin `out` in Figure 2–34, the software issues a warning that the Fitter cannot pack the node to an I/O pin because the node and the I/O cell are connected across a design partition boundary.

This type of register packing is not permitted because it would require moving logic across a design partition boundary to place into a single I/O device atom. To perform register packing, either the register must be moved out of the subdesign partition or the inverter must be moved into the subdesign partition to be implemented in the register. In order to guarantee correctness of the design with subsequent incremental compilations, the contents of design partitions must be preserved.

To allow the software to pack the register in the subdesign from Figure 2–32 with the input pin `in` in Figure 2–33 or the output pin `out` in Figure 2–34, make one of the following changes:

- Remove the design partition assignment from the subdesign. This allows the Fitter to perform all cross-hierarchy optimizations. However, it prevents you from using incremental compilation for this block of hierarchy. A good design partition should have a well-defined interface so that the Fitter does not have to perform cross-boundary optimizations.
- Restructure your HDL code to place the register in the same partition as the pin. The simplest option is to move the register from the subdesign partition into the top-level partition containing the pin. This ensures that the Fitter can optimize the two nodes without violating any partition boundaries.
- Restructure your HDL code so the register and the inverter are contained in the same partition. Move the inverter from the top-level block into the subdesign as shown in Figure 2–30 for an input pin. Then connect the subdesign to a pin in the top-level design, as shown in Figure 2–31 for an input pin.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF form.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Generate Incremental Compilation Tcl Script Command

To create a template Tcl script for full incremental compilation, use the Generate Incremental Compilation Tcl Script feature. Right-click in the **Design Partitions Window** and click **Generate Incremental Compilation Tcl Script**.

If you have made any partition assignments in the user interface, this script contains the Tcl equivalents of the assignments. The Tcl assignments are described in the following sections.

Preparing a Design for Incremental Compilation

To set or modify the current mode of incremental compilation, use the following command:

```
set_global_assignment -name INCREMENTAL_COMPILATION \
<value>
```

The incremental compilation <*value*> setting must be one of the following values:

- `FULL_INCREMENTAL_COMPILATION`—Full incremental compilation (this is the default)
- `INCREMENTAL_SYNTHESIS`—Incremental synthesis only
- `OFF`—No incremental compilation is performed

Creating Design Partitions

To create a partition, use the following command:

```
set_instance_assignment -name PARTITION_HIERARCHY \
<file name> -to <destination> -section_id <partition name>
```

The <*destination*> should be the entity's short hierarchy path. A short hierarchy path is the full hierarchy path without the top-level name (including quotation marks), for example:

```
"ram:ram_unit|altsyncram:altsyncram_component"
```

For the top-level partition, you can use the pipe (|) symbol to represent the top-level entity.



For more information about hierarchical naming conventions, refer to *Node-Naming Conventions in Quartus II Integrated Synthesis* in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

The <*partition name*> is the user-designated partition name, which must be unique and less than 1024 characters. The name can consist only of alphanumeric characters, and the pipe (|), colon (:), and underscore (_) characters. Altera recommends enclosing the name in double quotation marks (`" "`).

The <*file name*> is the name used for internally generated netlists files during incremental compilation. Netlists are named automatically by the Quartus II software based on the instance name if you create the partition in the user interface. If you are using Tcl to create your partitions, you must assign a custom file name that is unique across all partitions. For the top-level partition, the specified file name is ignored, and you can use any dummy value. To ensure the names are safe and platform independent, file names must be unique regardless of case. For example, if a partition uses the file name `my_file`, no other partition can use the file name `MY_FILE`. For simplicity, Altera recommends that you base each file name on the corresponding instance name for the partition.

The software stores all netlists in the **\db** compilation database directory.

Setting Properties of Design Partitions

After a partition is created, set its Netlist Type with the following command:

```
set_global_assignment -name PARTITION_NETLIST_TYPE <value> -section_id \
<partition name>
```

The netlist type <*value*> setting is one of the following values:

- `SOURCE`—Source File
- `POST_SYNTH`—Post-Synthesis
- `POST_FIT`—Post-Fit
- `STRICT_POST_FIT`—Post-Fit (Strict)
- `IMPORTED`—Imported
- `IMPORT_BASED_POST_FIT`—Post-Fit (Import-based)
- `EMPTY`—Empty

Set the Fitter Preservation Level for a post-fit or imported netlist using the following command:

```
set_global_assignment -name PARTITION_FITTER_PRESERVATION_LEVEL <value> \
-section_id <partition name>
```

The Fitter Preservation Level <*value*> setting is one of the following values:

- `NETLIST_ONLY`—Netlist only
- `PLACEMENT`—Placement
- `PLACEMENT_AND_ROUTING`—Placement and routing
- `PLACEMENT_AND_ROUTING_AND_TILE`— Placement and routing, as well as the power tile setting of high-speed or low-power

For details about these partition properties, refer to "Setting Properties of Design Partitions".

Creating Good Floorplan Location Assignments—Excluding or Filtering Certain Device Elements (Such as RAM or DSP Blocks)

Resource filtering uses the optional Tcl argument `-exclude_resources` in the `set_logiclock_contents` function of the LogicLock Tcl package. If left unspecified, no resource filter is created.

The argument takes a list of resources-to-be-excluded as input. The list is a colon-delimited string of the following keywords:

Table 2–4. Resources-to-be-Excluded Keywords

Keyword	Resource
REGISTER	Any registers in the logic cells
COMBINATIONAL	Any combinational elements in the logic cells
SMALL_MEM	The small TriMatrix memory blocks (M512 or MLAB)
MEDIUM_MEM	The medium TriMatrix memory blocks (M4K or M9K)
LARGE_MEM	The large TriMatrix memory blocks (M-RAM or M144K)
DSP	Any DSP blocks
VIRTUAL_PIN	Any virtual pins

For example, the following command assigns everything under **alu:alu_unit** to the ALU region, excluding all the DSP and M512 blocks:

```
set_logiclock_contents -region ALU -to alu:alu_unit -exceptions \
"DSP:SMALL_MEM"
```

In the QSF, resource filtering uses an extra LogicLock membership assignment called `LL_MEMBER_RESOURCE_EXCLUDE`. For example, the following line in the QSF is used to specify a resource filter for the **alu:alu_unit** entity assigned to the ALU region. The value of the assignment takes the same format as the resource listing string taken by the previous Tcl command.

```
set_instance_assignment -name LL_MEMBER_RESOURCE_EXCLUDE "DSP:SMALL_MEM" \
-to "alu:alu_unit" -section_id ALU
```

Generating Bottom-Up Design Partition Scripts

To generate scripts, type the following Tcl command at a Tcl prompt:

```
generate_bottom_up_scripts <options> ↵
```

The command is part of the `database_manager` package, which must be loaded using the following command before the command can be used:

```
load_package database_manager
```

You must open a project before you can generate scripts.

The Tcl options are the same as those available in the GUI. The exact format of each option is specified in Table 2–5.

Table 2–5. Options for Generating Bottom-Up Partition Scripts with Tcl Commands

Option	Default
`-include_makefiles` <*on*|*off*>	On
`-include_project_creation` <*on*|*off*>	On
`-include_virtual_pins` <*on*|*off*>	On
`-include_virtual_pin_timing` <*on*|*off*>	On
`-include_virtual_pin_locations` <*on*|*off*>	On
`-include_logiclock_regions` <*on*|*off*>	On
`-include_all_logiclock_regions` <*on*|*off*>	On
`-include_global_signal_promotion` <*on*|*off*>	Off
`-include_pin_locations` <*on*|*off*>	On
`-include_timing_assignments` <*on*|*off*>	On
`-include_design_partitions` <*on*|*off*>	On
`-remove_existing_regions` <*on*|*off*>	On
`-disable_auto_global_promotion` <*on*|*off*>	Off
`-bottom_up_scripts_output_directory` <*output directory*>	Current project directory
`-virtual_pin_delay` <*delay in ns*>	*(1)*

Note to Table 2–5:
(1) No default.

The following example shows how to use the Tcl command:

```
load_package database_manager
set project test_proj
```

```
project_open $project
generate_bottom_up_scripts -bottom_up_scripts_output_directory test \
    -include_virtual_pin_timing on -virtual_pin_delay 1.2
project_close
```

Command Line Support

To generate scripts at the command prompt, type the following command:

```
quartus_cdb <project name> --generate_bottom_up_scripts=on <options> ↵
```

Once again, the options map to the same as those in the GUI. To add an option, append "`--<option_name>=<val>`" to the command line call.

The command prompt options are the same as those available in the GUI. They are listed in Table 2–6.

Table 2–6. Options for Generating Bottom-Up Partition Scripts

Option	Default
`--include_makefiles_with_bottom_up_scripts=<on|off>`	On
`--include_project_creation_in_bottom_up_scripts=<on|off>`	On
`--include_virtual_pins_in_bottom_up_scripts=<on|off>`	On
`--include_virtual_pin_timing_in_bottom_up_scripts=<on|off>`	On
`--bottom_up_scripts_virtual_pin_delay=<delay in ns>`	(1)
`--include_virtual_pin_locations_in_bottom_up_scripts=<on|off>`	On
`--include_logiclock_regions_in_bottom_up_scripts=<on|off>`	On
`--include_all_logiclock_regions_in_bottom_up_scripts=<on|off>`	On
`--include_global_signal_promotion_in_bottom_up_scripts=<on|off>`	Off
`--include_pin_locations_in_bottom_up_scripts=<on|off>`	On
`--include_timing_assignments_in_bottom_up_scripts=<on|off>`	On
`--include_design_partitions_in_bottom_up_scripts=<on|off>`	On
`--remove_existing_regions_in_bottom_up_scripts=<on|off>`	On
`--disable_auto_global_promotion_in_bottom_up_scripts=<on|off>`	Off
`--bottom_up_scripts_output_directory=<output directory>`	Current project directory

Note to Table 2–6:
(1) No default. You must provide this option if you are including virtual pin timing.

Exporting a Partition to be Used in a Top-Level Project

Use the quartus_cdb executable to export a file for a bottom-up incremental compilation flow with the following command:

```
quartus_cdb --INCREMENTAL_COMPILATION_EXPORT=<file> \
[--incremental_compilation_export_netlist_type=<POST_SYNTH|POST_FIT>] \
[--incremental_compilation_export_partition_name=<partition name>] \
[--incremental_compilation_export_routing=<on|off>]
```

The <*file*> argument is the file path to the exported file. The <*partition name*> is the name of the partition, not its hierarchical path. If you do not specify the options, the executable uses any settings in the QSF file, or otherwise uses the default values. The default partition is the top-level partition in the project, the default netlist type is post-fit, and the default for routing is on (for all device families that support exported routing).

The command reads the assignment INCREMENTAL_COMPILATION_EXPORT_NETLIST_TYPE to determine which netlist type to export; the default is post-fit.

You can also use the flow INCREMENTAL_COMPILATION_EXPORT in the execute_flow Tcl command contained in the flow Tcl package.

Use the following commands to export a QXP file for a given partition, choose the netlist type, and specify whether to export routing.

```
load_package flow
set_global_assignment -name INCREMENTAL_COMPILATION_EXPORT_FILE <filename>
set_global_assignment -name INCREMENTAL_COMPILATION_EXPORT_NETLIST_TYPE \
<POST_FIT|POST_SYNTH>
set_global_assignment -name \
INCREMENTAL_COMPILATION_EXPORT_PARTITION_NAME <partition name>
set_global_assignment -name INCREMENTAL_COMPILATION_EXPORT_ROUTING \
<on|off>
execute_flow -INCREMENTAL_COMPILATION_EXPORT
```

The default partition is the top-level partition in the project, the default netlist type is post-fit, and the default for routing is on (for all device families that support exported routing).

To turn on the option to always perform exportation following compilation, use the following Tcl command:

```
set_global_assignment -name AUTO_EXPORT_INCREMENTAL_COMPILATION ON
```

Importing a Lower-Level Partition into the Top-Level Project

Use the quartus_cdb executable to import a lower-level partition with the following command:

```
quartus_cdb -- INCREMENTAL_COMPILATION_IMPORT ↵
```

You can also use the flow called INCREMENTAL_COMPILATION_IMPORT in the execute_flow Tcl command contained in the flow Tcl package.

The following example script shows how to import a partition using a Tcl script:

```
load_package flow
# commands to set the import-related assignments for each partition
execute_flow --INCREMENTAL_COMPILATION_IMPORT
```

Specify the location for the imported file with the PARTITION_IMPORT_FILE assignment. Note that the file specified by this assignment is read only during importation. For example, the project is completely independent from any files from the lower-level projects after importing. In the command-line and Tcl flow, any partition that has this assignment set to a non-empty value will be imported.

The following assignments specify how the partition should be imported:

```
PARTITION_IMPORT_PROMOTE_ASSIGNMENTS = on | off
PARTITION_IMPORT_NEW_ASSIGNMENTS = on | off
PARTITION_IMPORT_EXISTING_ASSIGNMENTS = \
replace_conflicting | skip_conflicting
PARTITION_IMPORT_EXISTING_LOGICLOCK_REGIONS = \
replace_conflicting | update_conflicting | skip_conflicting
```

Makefiles

For an example of how to use incremental compilation with a makefile as part of the bottom-up design flow, refer to the **read_me.txt** file that accompanies the incr_comp example located in the **/qdesigns/incr_comp_makefile** subdirectory. When using a bottom-up incremental compilation flow, the Generate Bottom-Up Design Partition Scripts feature can write makefiles that automatically export lower-level design partitions and import them into the top-level project whenever design files change.

Recommended Design Flows and Compilation Application Examples

This section provides scripting examples that cover some of the topics discussed in the main section of the chapter.

The script shown in Example 2–1 opens a project called AB_project, sets up two partitions, entities **A** and **B**, for the first time, and performs an initial complete compilation.

Example 2–1. AB_project

```
set project AB_project

package require ::quartus::flow
project_open $project

# Ensure that incremental compilation is turned on
set_global_assignment -name INCREMENTAL_COMPILATION \
FULL_INCREMENTAL_COMPILATION

# Set up the partitions
set_instance_assignment -name PARTITION_HIERARCHY \
 db/A_inst -to A -section_id "Partition_A"
set_instance_assignment -name PARTITION_HIERARCHY \
 db/B_inst -to B -section_id "Partition_B"

# Set the netlist types to post-fit for subsequent
# compilations (all partitions are compiled during the
# initial compilation since there are no post-fit
# netlists)
set_global_assignment -name PARTITION_NETLIST_TYPE \
 POST_FIT -section_id "Partition_A"
set_global_assignment -name PARTITION_NETLIST_TYPE \
 POST_FIT -section_id "Partition_B"

# Run initial compilation:
export_assignments
execute_flow -full_compile

project_close
```

Design Flow 1—Changing a Source File for One of Multiple Partitions in a Top-Down Compilation Flow

Example background: You have run the initial compilation shown in the example script under "Recommended Design Flows and Compilation Application Examples" on page 2–62. You have modified the HDL source file for partition **A**, and would like to recompile it.

Run the standard flow compilation command in your Tcl script:

```
execute_flow -full_compile
```

Or, run the following command at a system command prompt:

```
quartus_sh --flow compile AB_project↵
```

Assuming the source files for partition **B** do not depend on **A**, only **A** is recompiled. The placement of **B** and its timing performance is preserved, which also saves significant compilation time.

Design Flow 2—Optimizing the Placement for One of Multiple Partitions in a Top-Down Compilation Flow

Example background: You have run the initial compilation shown in the example script under "Recommended Design Flows and Compilation Application Examples" on page 2–62. You would like to apply Fitter optimizations, such as physical synthesis, only to partition **A**. No changes have been made to the HDL files.

To ensure the previous compilation result for partition **B** is preserved, and to ensure that Fitter optimizations are applied to the post-synthesis netlist of partition **A**, set the netlist type of **B** to Post-Fit (which was already done in the initial compilation, but is repeated here for safety), and the netlist type of **A** to Post-Synthesis, as shown in the following script:

```
set project AB_project

package require ::quartus::flow
project_open $project

# Turn on Physical Synthesis Optimization
set_global_assignment -name \
PHYSICAL_SYNTHESIS_REGISTER_RETIMING ON

# For A, set the netlist type to post-synthesis
set_global_assignment -name PARTITION_NETLIST_TYPE POST_SYNTH \
-section_id "Partition_A"

# For B, set the netlist type to post-fit
set_global_assignment -name PARTITION_NETLIST_TYPE POST_FIT \
-section_id "Partition_B"

# Run incremental compilation:
export_assignments
execute_flow -full_compile

project_close
```

Conclusion

With the Quartus II incremental compilation feature described in this chapter, you can preserve the results and the performance of unchanged logic in your design as you make changes elsewhere. The various applications of incremental compilation enable you to improve your productivity while designing for high-density FPGAs, using either top-down or bottom-up design methodologies.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*
- *Introduction to Quartus II Manual*
- *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*

- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Switching to the TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Synthesis* section in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 2–7 shows the revision history for this chapter.

Table 2–7. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Updated "Introduction" on page 2–1. • Updated "Choosing a Quartus II Compilation Flow" on page 2–3. • Changed title and updated "Preparing a Design for Incremental Compilation" section to "Quick Start Guide – Summary of Steps for an Incremental Compilation Flow" on page 2–11. • Updated "Design Partition Assignments Compared to Physical Placement Assignments" on page 2–18. • Updated "Creating Design Partitions" on page 2–19. • Updated "Creating a Design Floorplan With LogicLock Location Assignments" on page 2–29. • Updated "Exporting and Importing Partitions for Bottom-Up Design Flows" on page 2–32. • Updated "Guidelines for Creating Good Design Partitions and LogicLock Regions" on page 2–46. • Updated "Incremental Compilation Restrictions" on page 2–76.	Updated for Quartus II software version 7.2.
May 2007 v7.1.0	• Updated "Choosing a Quartus II Compilation Flow" on page 2–3.Updated "Preparing a Design for Incremental Compilation" on page 2–10. • Updated Tables 2–1 and 2–3. • Updated design in "Recommended Design Flows and Compilation Application Examples" on page 2–61. • Added new examples to "Design Flow 7—Creating Hard-Wired Macros for IP Reuse" on page 2–72. • Moved and simplified "Using Incremental Synthesis Only Instead of Full Incremental Compilation" on page 2–76. • Updated "HardCopy Compilation Flows" on page 2–81. • Updated "Support for the TimeQuest Timing Analyzer and SDC Constraints" on page 2–81. • Updated "Setting Properties of Design Partitions" on page 2–98. • Added "Referenced Documents" on page 2–106.	Removed several dialog box figures. Added support for Arria GX devices. Added Fitter Preservation Level Post-Fit Placement, Routing, and Tiles.
March 2007 v7.0.0	No changes to chapter.	—

Table 2–7. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
November 2006 v6.1.0	Chapter 2 was formerly Chapter 1 in version 6.0.0. Reorganized chapter to group recommendations and guidelines together. Updated for the Quartus II software version 6.1: • Added support for Stratix III devices. • Added information on the Incremental Compilation Advisor. • The full incremental compilation option is now turned on by default. • Added new feature for Exporting a Lower-Level Block within a Project. • Changed the location of the **Automatically export design partition after compilation** option. • Added support for HardCopy Compilation Flows. • Added that routing can be exported in bottom-up flows. • Added I/O port guidelines in Creating Good Design Partitions. • Updated limitations: SignalProbe Pins and Engineering Change Management with the Chip Planner.	Added support for Stratix III devices. Added information about new features and updates in the Quartus II software version 6.1.
May 2006 v6.0.0	Name changed to *Quartus II Incremental Compilation for Hierarchical and Team-Based Design*. Updated for the Quartus II software version 6.0. • Added new device support information. • Added top-down and bottom-up design flow information. • Added incremental compilation design compiling information. • Added recommendations for creating good floorplan location assignments. • Added register packing and partition boundary information. • Added engineering management with the Chip Editor. • Added information on how to check and save to reapply SignalProbe. • Added user scenarios.	—
December 2005 v5.1.1	Minor typographic update.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
August 2005 v5.0.1	Added documentation on cross-partition register packing.	—
May 2005 v5.0.0	Initial release.	—

3. Quartus II Design Flow for MAX+PLUS II Users

QII51002-7.2.0

Introduction

The feature-rich Quartus® II software helps you shorten your design cycles and reduce time-to-market. With support for FLEX®, ACEX®, and MAX® device families, as well as all of Altera®'s newest devices, the Quartus II software is the most widely accepted Altera design software tool today.

This chapter describes how to convert MAX+PLUS® II designs to Quartus II projects, as well as the similarities and differences between the MAX+PLUS II and Quartus II design flows. This discussion includes supported device families, graphical user interface (GUI) comparisons, and the advantages of the Quartus II software.

There are many features in the Quartus II software to help MAX+PLUS II users easily transition to the Quartus II software design environment. These include a customizable **Look & Feel** feature, which changes the GUI to display menus, toolbars, and utility windows as they appear in the MAX+PLUS II software without sacrificing Quartus II software functionality.

Chapter Overview

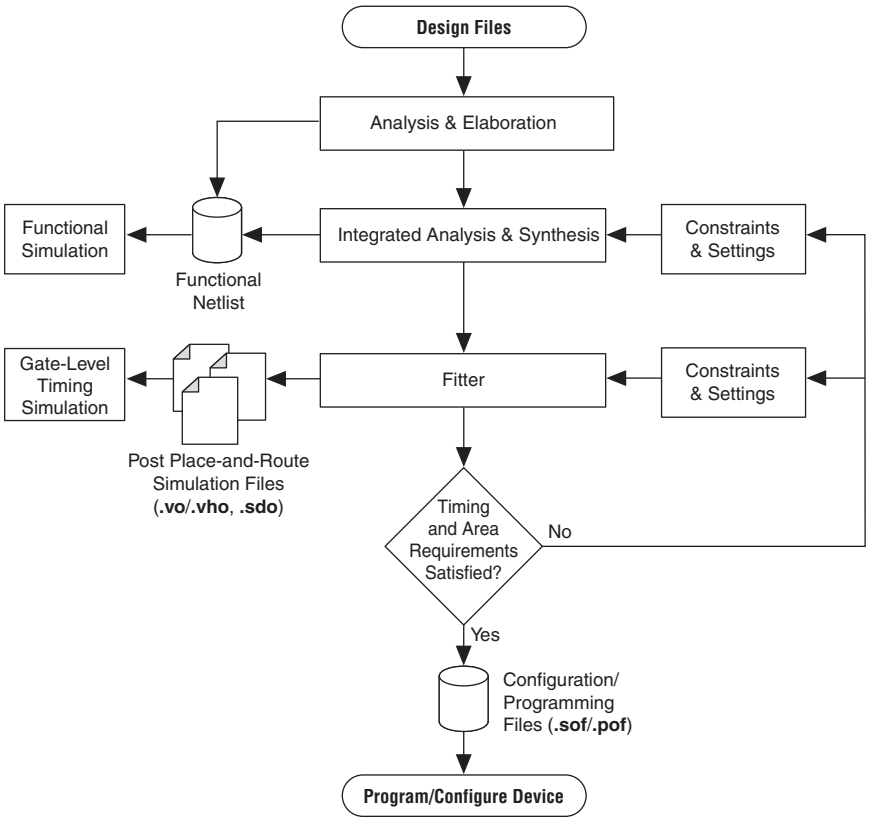
This chapter covers the following topics:

- "Typical Design Flow" on page 3–2
- "Device Support" on page 3–3
- "Quartus II GUI Overview" on page 3–4
- "Setting Up MAX+PLUS II Look and Feel in Quartus II" on page 3–6
- "Compiler Tool" on page 3–9
- "MAX+PLUS II Design Conversion" on page 3–12
- "Quartus II Design Flow" on page 3–15
- "Quick Menu Reference" on page 3–35

Typical Design Flow

Figure 3–1 shows a typical design flow with the Quartus II software.

Figure 3–1. Quartus II Software Design Flow



Device Support

The Quartus II software supports most of the devices supported in the MAX+PLUS II software, but it does not support any obsolete devices or packages. The devices supported by these two software packages are shown in Table 3–1.

Table 3–1. Device Support Comparison

Device Supported	Quartus II	MAX+PLUS II
Arria GX™	✓	—
Stratix® Series	✓	—
Cyclone® Series	✓	—
Hardcopy® Series	✓	—
MAX® II	✓	—
Classic™	—	✓
MAX 3000A	✓	✓
MAX 7000S/AE/B	✓	✓
MAX 7000E	—	✓
MAX 9000	—	✓
ACEX® 1K	✓	✓
FLEX® 6000	✓	✓
FLEX 8000	—	✓
FLEX 10K	✓ *(1)*	✓
FLEX 10KA	✓	✓
FLEX 10KE	✓ *(2)*	✓
Mercury™	✓	—
APEX™ II	✓	—
APEX™ 20K	✓	—

Notes to Table 3–1:

(1) PGA packages (represented as package type G in the ordering code) are not supported in the Quartus II software.

(2) Some packages are not supported.

Quartus II GUI Overview

The Quartus II software provides the following utility windows to assist in the development of your designs:

- Project Navigator
- Node Finder
- Tcl Console
- Messages
- Status
- Change Manager

Project Navigator

The **Hierarchy** tab of the Project Navigator window is similar to the MAX+PLUS II Hierarchy Display and provides additional information such as logic cell, register, and memory bit resource utilization. The **Files** and **Design Units** tabs of the Project Navigator window provide a list of project files and design units.

Node Finder

The Node Finder window provides the equivalent functionality of the MAX+PLUS II **Search Node Database** dialog box and allows you to find and use any node name stored in the project database.

Tcl Console

The Tcl Console window allows access to the Quartus II Tcl shell from within the GUI. You can use the Tcl Console window to enter Tcl commands and source Tcl scripts to make assignments, perform customized timing analysis, view information about devices, or fully automate and customize the way you run all components of the Quartus II software. There is no equivalent functionality in the MAX+PLUS II software.



For more information on using Tcl with the Quartus II software, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Messages

The Messages window is similar to the Message Processor window in the MAX+PLUS II software, providing detailed information, warnings, and error messages.You also can use it to locate a node from a message to various windows in the Quartus II software.

Status

The Status window displays information similar to the MAX+PLUS II Compiler window. Progress and elapsed time are shown for each stage of the compilation.

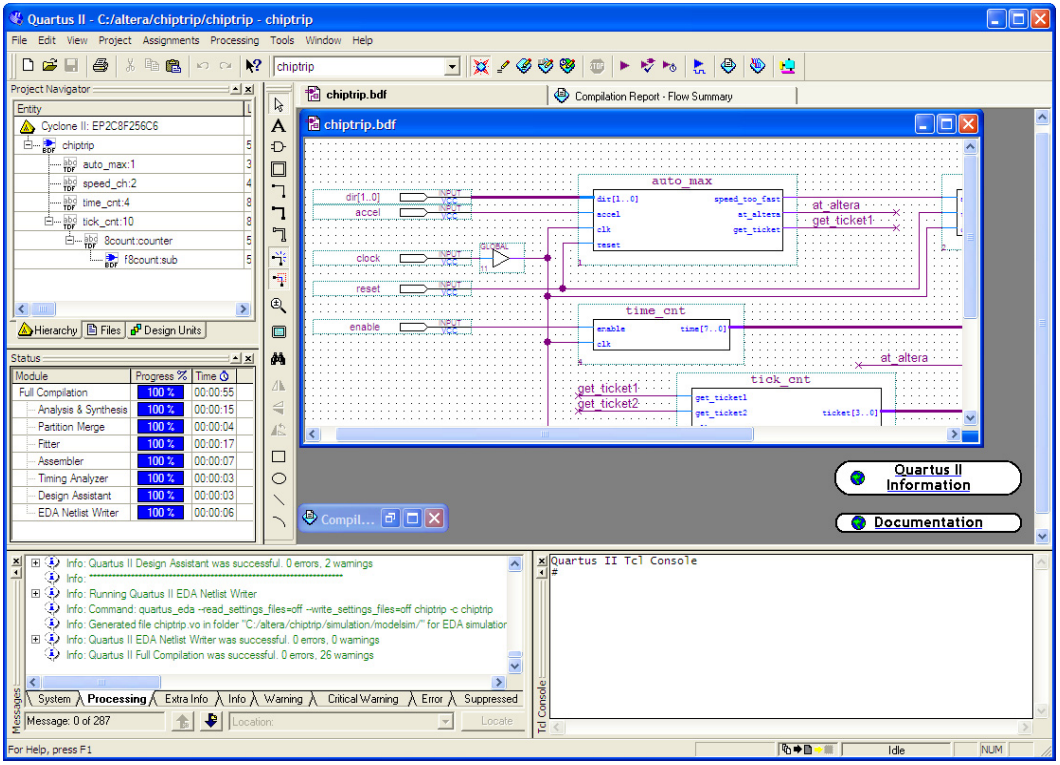
Change Manager

The Change Manager provides detailed tracking information on all design changes made with the Chip Planner.

For more information about the Engineering Change Manager and the Chip Editor, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Figure 3–2 shows a typical Quartus II software display.

Figure 3–2. Quartus II Look and Feel



Setting Up MAX+PLUS II Look and Feel in Quartus II

You can choose the MAX+PLUS II look and feel by selecting MAX+PLUS II in the **Look & Feel** box of the **General** tab of the **Customize** dialog box on the Tools menu.



Any changes to the look and feel do not become effective until you restart the Quartus II software.

By default, when you select the MAX+PLUS II look and feel, the **MAX+PLUS II** quick menu (Figure 3–21 on page 3–35) appears on the left side of the menu bar. You can turn the Quartus II and MAX+PLUS II quick menus on or off. You also can change the preferred positions of the two quick menus. To change these options, perform the following steps:

1. On the Tools menu, click **Customize**. The **Customize** dialog box is shown.
2. Click the **General** tab.
3. Under **Quick menus**, select your preferred options.

MAX+PLUS II Look and Feel

The MAX+PLUS II look and feel in the Quartus II software closely resembles the MAX+PLUS II software. Figures 3–3 and 3–4 compare the MAX+PLUS II software appearance with the Quartus II MAX+PLUS II look and feel.

Figure 3–3. MAX+PLUS II Software GUI

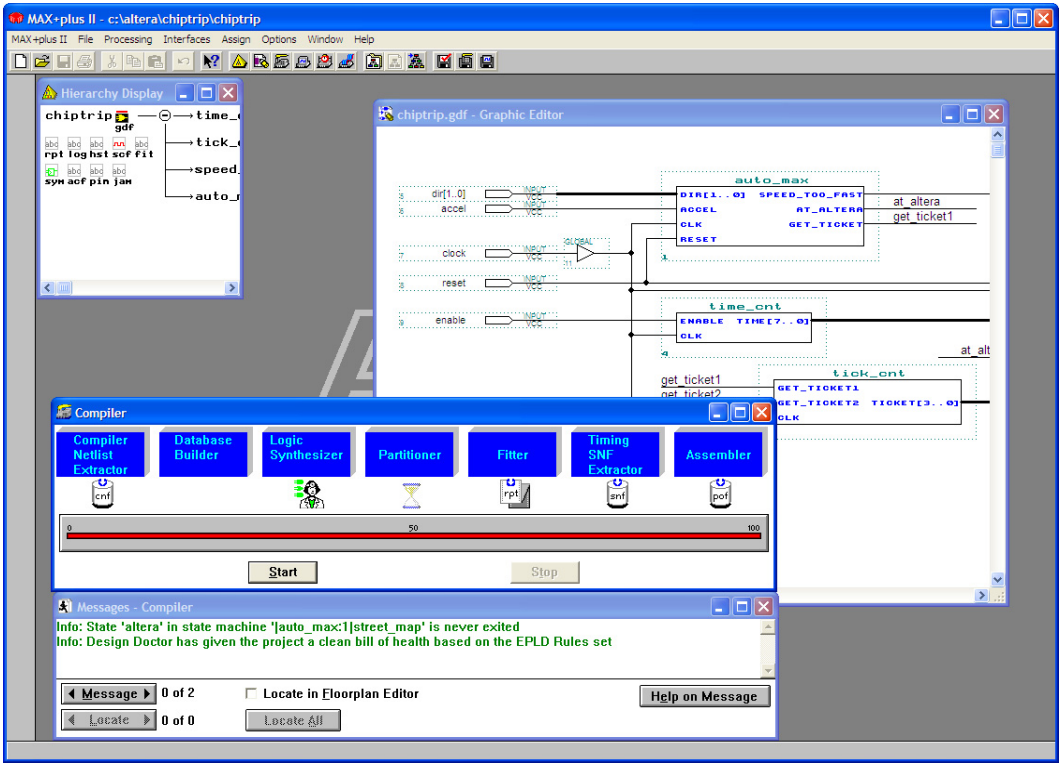
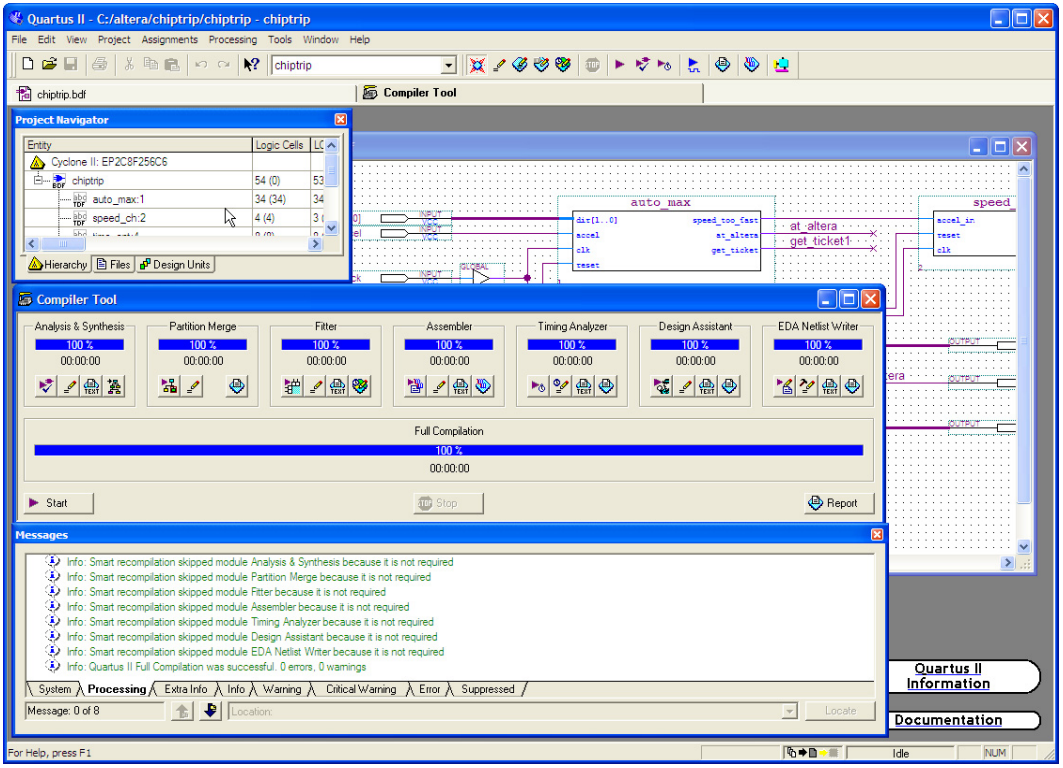


Figure 3–4. Quartus II Software with MAX+PLUS II Look and Feel



The standard MAX+PLUS II toolbar is also available in the Quartus II software with the MAX+PLUS II look and feel in the Quartus II software (Figure 3–5).

Figure 3–5. Standard MAX+PLUS II Toolbar



Compiler Tool

The Quartus II Compiler Tool provides an intuitive MAX+PLUS II style interface. You can edit the settings and view result files for the following modules:

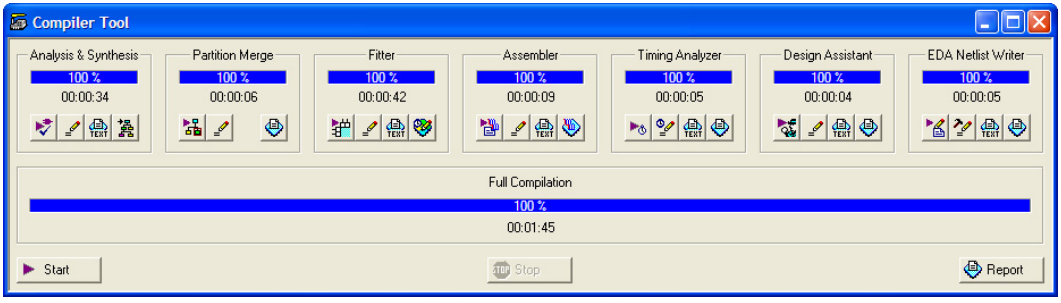
- Analysis and Synthesis
- Partition Merge
- Fitter
- Assembler
- Timing Analyzer
- EDA Netlist Writer
- Design Assistant

Each of these modules is described later in this section.

To start a compilation using the Compiler Tool, click **Compiler Tool** from either the MAX+PLUS II menu or the Tools menu and click **Start** in the Compiler Tool. The Compiler Tool, shown in Figure 3–6, displays all modules, including optional modules such as Partition Merge, Assembler, EDA Netlist Writer, and the Design Assistant.

For information about using the Quartus II software modules at the command line, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Figure 3–6. Running a Full Compilation with the Compiler Tool



Analysis and Synthesis

The Quartus II Analysis and Synthesis module analyzes your design, builds the design database, optimizes the design for the targeted architecture, and maps the technology to the design logic.

In MAX+PLUS II software, these functions are performed by the Compiler Netlist Extractor, Database Builder, and Logic Synthesizer. There is no module in the Quartus II software similar to the MAX+PLUS II Partitioner module.

Partition Merge

The optional Quartus II Partition Merge module merges the partitions to create a flattened netlist for further stages of the Quartus II compilation flow. The Partition Merge module is not similar to the MAX+PLUS II Partitioner. This tool is available only if you turn on incremental compilation. You can turn on incremental compilation by performing the following steps:

1. On the Assignment menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the **+** icon to expand **Compilation Process Settings**, and select **Incremental Compilation**. The **Full Incremental Compilation** page appears.
3. Under **Incremental compilation**, turn on Incremental Compilation.

Fitter

The Quartus II Fitter module uses the PowerFit™ fitter to fit your design into the available resources of the targeted device. The Fitter places and routes the design. The Fitter module is similar to the Fitter stage of the MAX+PLUS II software.

Assembler

The optional Quartus II Assembler module creates a device programming image of your design so that you can configure your device. You can select from the following types of programming images:

- Programmer Object File (**.pof**)
- SRAM Output File (**.sof)**
- Hexadecimal (Intel-Format) Output File (**.hexout**)
- Tabular Text File (**.ttf)**
- Raw Binary File (**.rbf**)
- Jam™ STAPL Byte Code 2.0 File (**.jbc)**
- JEDEC STAPL Format File (**.jam**)

You can turn off the Assembler module during compilation by turning off **Run assembler** in the **Compilation Process Settings** page in the **Settings** dialog box. You also can turn off the Assembler by right-clicking in the Compiler Tool window. The Assembler module is similar to the Assembler stage of the MAX+PLUS II software.

Timing Analyzer

The Quartus II Timing Analyzer allows you to analyze more complex clocking schemes than is possible with the MAX+PLUS II Timing Analyzer. The Quartus II Timing Analyzer analyzes all clock domains in your design, including paths that cross clock domains, and also reports both f_{MAX} and slack. Slack is the margin by which the timing requirement is met or is not met. For more information on the Timing Analyzer, refer to "Timing Analysis" on page 3–27.

EDA Netlist Writer

The optional Quartus II EDA Netlist Writer module generates a netlist for simulation with an EDA simulation tool. The EDA Netlist Writer module is comparable to the VHDL and Verilog Netlist Writer in the MAX+PLUS II software.

Design Assistant

The optional Quartus II Design Assistant module checks the reliability of your design based on a set of design rules. The Design Assistant analyzes and generates messages for a design targeting any Altera device and is especially useful for checking the reliability of a design to be converted to HardCopy series devices. The Design Assistant is similar to the Design Doctor in the MAX+PLUS II software.

In the Quartus II software, you can reduce subsequent compilation time significantly by turning **Use Smart compilation** on before compiling your design. The Smart Compilation feature skips any compilation stages which are not required and which may use more disk space. This Quartus II smart compilation option is similar to the MAX+PLUS II **Smart Recompile** command. To turn the **Use Smart compilation** option on, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Compilation Process Settings**. The **Compilation Process Settings** page appears.
3. Turn on **Use Smart compilation**.

MAX+PLUS II Design Conversion

With the Quartus II software, you can open MAX+PLUS II designs and convert MAX+PLUS II assignments and files.

The Quartus II software is project based. All the files for your design (HDL input, simulation vectors, assignments, and other relevant files) are associated with a project file. For more information about creating a new project, refer to "Creating a New Project" on page 3–16.

Converting an Existing MAX+PLUS II Design

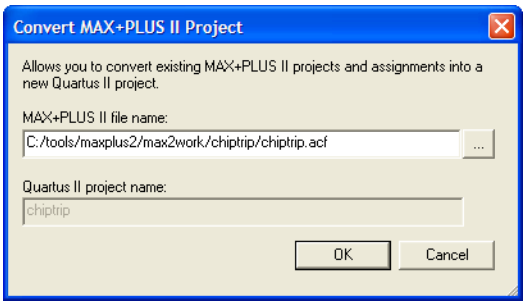
You can easily convert an existing MAX+PLUS II design for use with the Quartus II software with the **Convert MAX+PLUS II Project** command in the Quartus II software or the **Open Project** command. You can find these commands on the File menu

If you use the **Convert MAX+PLUS II Project** command, browse to the MAX+PLUS II Assignments and Configuration File (**.acf**) or top-level design file (Figure 3–7) and click **Open**. The **Convert MAX+PLUS II Project** command generates a Quartus II Project File (**.qpf**) and a Quartus II Settings File (**.qsf**). The Quartus II software stores project and design assignments in the Quartus II Settings File, which is equivalent to the Assignments and Configuration File in the MAX+PLUS II software.

You also can open and convert a MAX+PLUS II design with the **Open Project** command. In the **Open Project** dialog box, browse to the Assignments and Configuration File or the top-level design file. Click **Open** to display the **Convert MAX+PLUS II Project** dialog box.

The Quartus II software can import all MAX+PLUS II-generated files, but it cannot save files in the MAX+PLUS II format. You cannot open a Quartus II project in the MAX+PLUS II software, nor can you convert a Quartus II project to a MAX+PLUS II project.

Figure 3–7. Convert MAX+PLUS II Project Dialog Box



The conversion process performs the following actions:

- Converts the MAX+PLUS II Assignments and Configuration File into a Quartus II Settings File (equivalent to importing all MAX+PLUS II assignments)
- Creates a Quartus II Project File
- Displays all errors and warnings in the Quartus II message window

The Quartus II software can read MAX+PLUS II generated Graphic Design Files (**.gdf**) and Simulation Channel Files (**.scf**) without converting them. These files are not modified during a MAX+PLUS II design conversion.

Converting MAX+PLUS II Graphic Design Files

The Quartus II Block Editor (similar to the MAX+PLUS II Graphic Editor) saves files as Block Design Files (**.bdf**). You can convert your MAX+PLUS II Graphic Design File into a Quartus II Block Design File using one of the following methods:

1. Open the Graphic Design File and on the File menu, click **Save As**. The **Save As** dialog box is shown.
2. In the **Save as type** list, select **Block Diagram/Schematic File** (*.**bdf**).

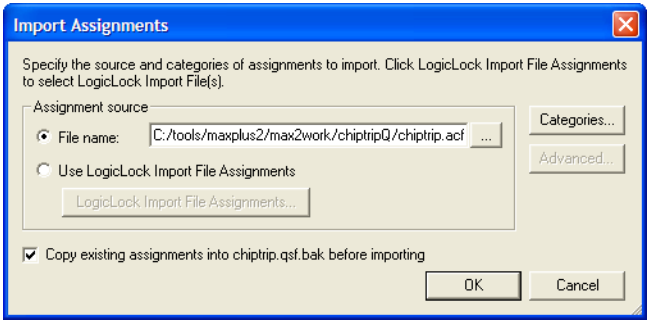
3. Run the **quartus_g2b.exe** command line executable located in the \<*Quartus II installation*>**bin** directory. For example, to convert the **chiptrip.gdf** file to a Block Design File, type the following command at a command prompt:

```
quartus_g2b.exe chip_trip.gdf ↵
```

Importing MAX+PLUS II Assignments

You can import MAX+PLUS II assignments into an existing Quartus II project. Open the project, and on the Assignments menu, click **Import Assignments**. Browse to the Assignments and Configuration File (Figure 3–8). You can also import Quartus II Settings Files and Entity Setting Files (**.esf**).

Figure 3–8. Import Assignments Dialog Box



The Quartus II software accepts most MAX+PLUS II assignments. However, some assignments can be imported incorrectly from the MAX+ PLUS II software into the Quartus II software due to differences in node naming conventions and the advanced Quartus II integrated synthesis algorithms.

The differing node naming conventions in the Quartus II and MAX+PLUS II software can cause improper mapping when importing your design from MAX+PLUS II software into the Quartus II software. Improper node names can interfere with the design logic if you are unaware of these node name differences and do not take appropriate

steps to prevent improper node name mapping. Table 3–2 compares the differences between the naming conventions used by the Quartus II and MAX+PLUS II software.

Table 3–2. Quartus II and MAX+PLUS II Node and Pin Naming Conventions

Feature	Quartus II Format	MAX+PLUS II Format
Node name	`auto_max:auto|q0`	`|auto_max:auto|q0`
Pin name	`d[0], d[1], d[2]`	`d0, d1, d2`

When you import MAX+PLUS II assignments containing node names that use numbers, such as `signal0` or `signal1`, the Quartus II software imports the original assignment and also creates an additional copy of the assignment. The additional assignment has square brackets inserted around the number, resulting in `signal[0]` or `signal[1]`. The square bracket format is legal for signals that are part of a bus, but creates illegal signal names for signals that are not part of a bus in the Quartus II software. If your MAX+PLUS II design contains node names that end in a number and are not part of a bus, you can edit the Quartus II Settings File to remove the square brackets from the node names after importing them.



You can remove obsolete assignments in the **Remove Assignments** dialog box. Open this dialog box on the Assignments menu by clicking **Remove Assignments**.

The Quartus II software may not recognize valid MAX+PLUS II node names, or may split MAX+PLUS II nodes into two different nodes. As a result, any assignments made to synthesized nodes are not recognized during compilation.



For more information about Quartus II node naming conventions, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Quartus II Design Flow

The following sections include information to help you get started using the Quartus II software. They describe the similarities and differences between the Quartus II software and the MAX+PLUS II software. The following sections highlight improvements and benefits in the Quartus II software.



For an overview of the Quartus II software features and design flow, refer to the *Introduction to Quartus II* manual.

Creating a New Project

The Quartus II software provides a wizard to help you create new projects. On the File menu, click **New Project Wizard** to start the New Project Wizard. The New Project Wizard generates the Quartus II Project File and Quartus II Settings File for your project.

Design Entry

The Quartus II software supports the following design entry methods:

- Altera HDL (AHDL) Text Design File (.**tdf**)
- Block Diagram File
- EDIF Netlist File (.**edf**)
- Verilog Quartus Mapping Netlist File (.**vqm**)
- VHDL (.**vhd**)
- Verilog HDL (.**v**)

The Quartus II software has an advanced integrated synthesis engine that fully supports the Verilog HDL and VHDL languages and provides options to control the synthesis process.

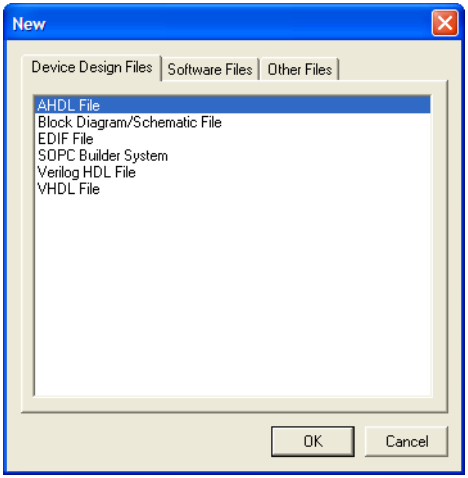


For more information, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

To create a new design file, perform the following steps:

1. On the File menu, click **New**. The **New** dialog box appears.
2. Click the **Device Design Files** tab.
3. Select a design entry type.
4. Click **OK** (see Figure 3–9).

Figure 3–9. New Dialog Box



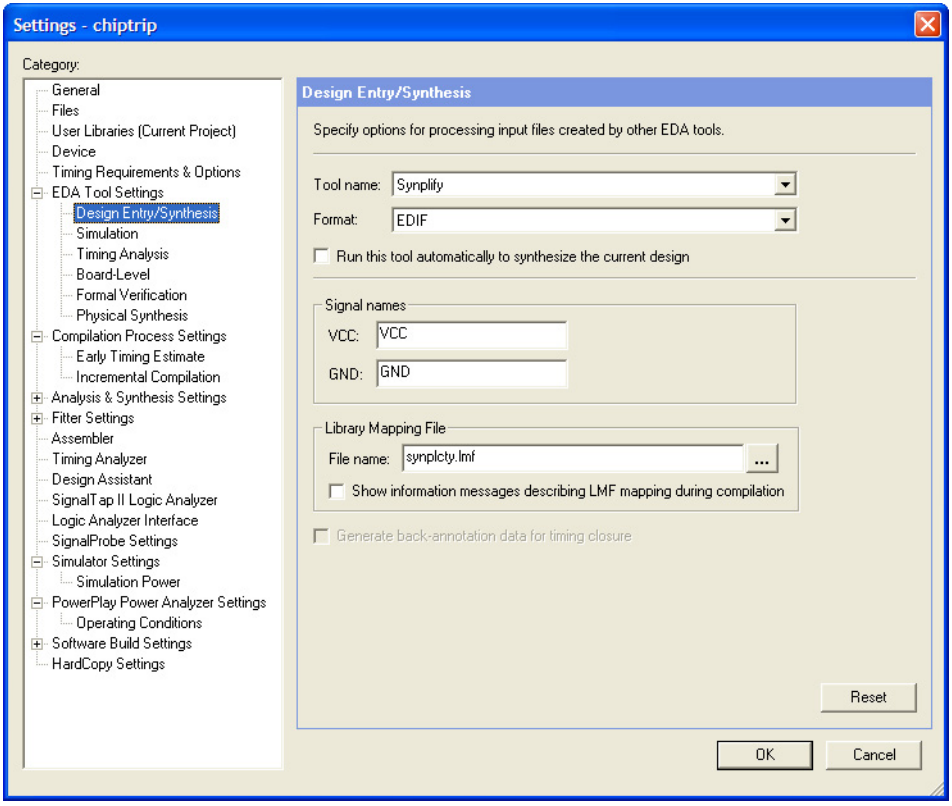


You can create other files from the **Software Files** tab and **Other Files** tab of the **New** dialog box on the File menu. For example, the Vector Waveform File (**.vwf**) is located in the **Other Files** tab.

To analyze a netlist file created by an EDA tool, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Design Entry & Synthesis**. The **Design Entry & Synthesis** page appears.
3. In the **Tool** name list, select the synthesis tool used to generate the netlist (Figure 3–10).

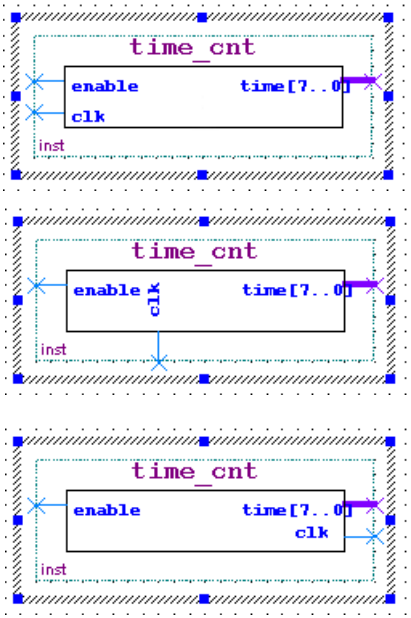
Figure 3–10. Settings Dialog Box Specifying Design Entry Tool



The Quartus II Block Editor has many advantages over the MAX+PLUS II Graphic Editor. The Block Editor offers an unlimited sheet size, multiple region selections, an enhanced Symbol Editor, and conduits.

The Symbol Editor allows you to change the positions of the ports in a symbol (refer to the three images in Figure 3–11). You can reduce wire congestion around a symbol by changing the positions of the ports.

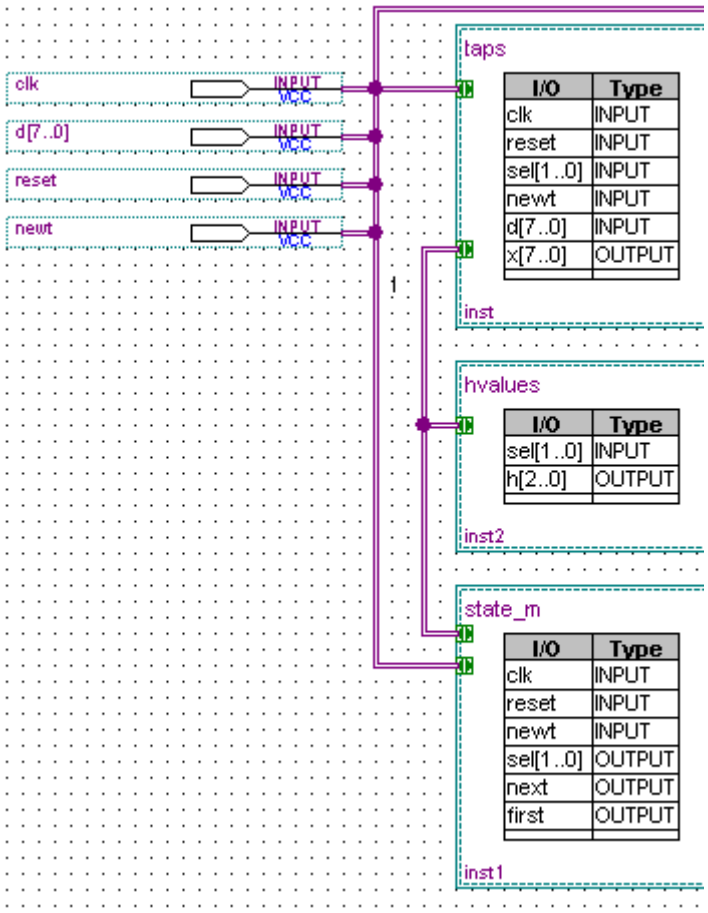
Figure 3–11. Various Port Position for a Symbol



To make changes to a symbol in a Block Design File, right-click a symbol in the Block Editor and select **Properties** to display the **Symbol Properties** dialog box. This dialog box allows you to change the instance name, add parameters, and specify the line and text color.

You can use conduits to connect blocks (including pins) in the Block Editor. Conduits contain signals for the connected objects (see Figure 3–12). You can determine the connections between various blocks in the **Conduit Properties** dialog box by right-clicking a conduit and clicking **Properties**.

Figure 3–12. Blocks and Pins Connected with Conduits



Making Assignments

The Quartus II software stores all project and design assignments in a Quartus II Settings File, which is a collection of assignments stored as Tcl commands and organized by the compilation stage and assignment type. The Quartus II Settings File stores all assignments, regardless of how they are made, from the Floorplan Editor, the Pin Planner, the Assignment Editor, with Tcl, or any other method.

Assignment Editor

The Assignment Editor is an intuitive spreadsheet interface designed to allow you to make, change, and manage a large number of assignments easily. With the Assignment Editor, you can list all available pin numbers and design pin names for efficiently creating pin assignments. You also can filter all assignments based on assignment categories and node names for viewing and creating assignments.

The Assignment Editor is composed of the Category Bar, Node Filter Bar, Information Bar, Edit Bar, and spreadsheet.

To make an assignment, follow these steps:

1. On the Assignments menu, click **Assignment Editor**. The **Assignment Editor** window appears.
2. Select an assignment category in the **Category** bar.
3. Select a node name using the Node Finder or type a node name filter into the **Node Filter** bar. (This step is optional; it excludes all assignments unrelated to the node name.)
4. Type the required values into the spreadsheet.
5. On the File menu, click **Save**.

If you are unsure about the purpose of a cell in the spreadsheet, select the cell and read the description displayed in the **Information** bar.

You can use the **Edit** bar to change the contents of multiple selected cells simultaneously. Select cells in the spreadsheet and type the value in the **Edit** box.

Other advantages of the Assignment Editor include clipboard support in the spreadsheet and automatic font coloring to identify the status of assignments.



For more information, refer to the *Assignment Editor* chapter in volume 1 of the *Quartus II Handbook.*

Timing Assignments

You can use the timing wizard to help you set your timing requirements. On the Assignments menu, click **Timing Wizard** to create global clock and timing settings. The settings include f_{MAX}, setup times, hold times, clock to output delay times, and individual absolute or derived clocks.

You also can set timing settings manually by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Setting** dialog box is shown.
2. In the **Category** list, select **Timing Requirements & Options**. The **Timing Requirements & Options** page is shown.
3. Set your timing settings.

You can make more complex timing assignments with the Quartus II software than allowed by the MAX+PLUS II software, including multicycle and point-to-point assignments using wildcards and time groups.



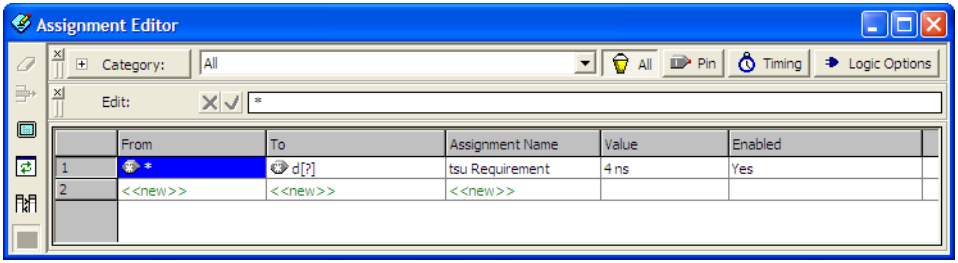
A time group is a collection of design nodes grouped together and represented as a single unit for the purpose of making timing assignments to the collection.

Multicycle timing assignments allow you to identify register-to-register paths in the design where you expect a delayed latch edge. This assignment enables accurate timing analysis of your design.

Point-to-point timing assignments allow you to specify the required delay between two pins, two registers, or a pin and a register. This assignment helps you optimize and verify your design timing requirements.

Wildcard characters "?" and " * " allow you to apply an assignment to a large number of nodes with just a few assignments. For example, Figure 3–13 shows a 4 ns t_{SU} requirement assignment to all paths from any node to the "d" bus in the Assignment Editor.

Figure 3–13. Single t_{SU} Timing Assignment Applied to All Nodes of a Bus



For more information, refer to the *Classic Timing Analyzer* or the *TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Synthesis

The Quartus II advanced integrated synthesis software fully supports the hardware description languages, Verilog HDL, VHDL, and AHDL, schematic entry, and also provides options to control the synthesis process. With this synthesis support, the Quartus II software provides a complete, easy-to-use, stand-alone solution for today's designs.

You can specify synthesis options in the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Similar to MAX+PLUS II synthesis options, you select one of these optimization techniques: **Speed, Area,** or **Balanced**.

To achieve higher design performance, you can turn on synthesis netlist optimizations that are available when targeting certain devices. You can unmap a netlist created by an EDA tool and remap the components in the netlist back to Altera primitives by turning on **Perform WYSIWYG primitive resynthesis**. Additionally, you can move registers across combinational logic to balance timing without changing design functionality by turning on **Perform gate-level register retiming**. Both of these options are accessible from the **Synthesis Netlist Optimizations** page under **Analysis & Synthesis Settings** in the **Settings** dialog box on the Assignments menu.

For more information, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Functional Simulation

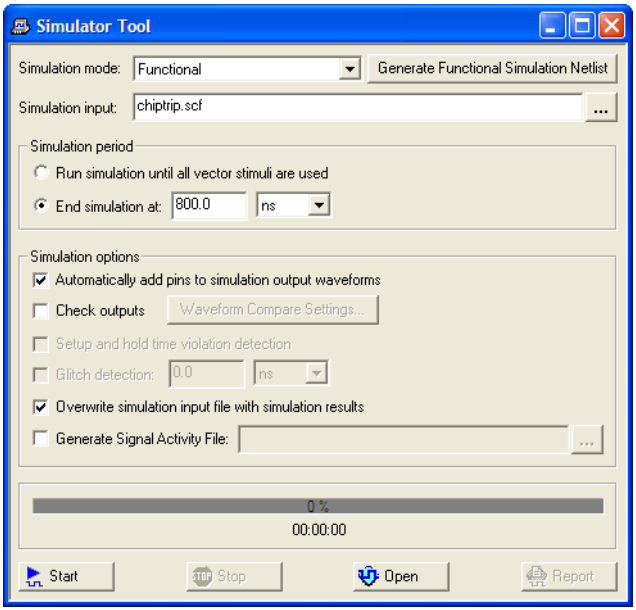
Similar to the MAX+PLUS II Simulator, the Quartus II Simulator Tool performs both functional and timing simulations.

To open the Simulator Tool, on MAX+PLUS II menu, click **Simulator** or on the Tools menu, click **Simulator Tool**. Before you perform a functional simulation, an internal functional simulation netlist is required. Click **Generate Functional Simulation Netlist** in the **Simulator Tool** window (Figure 3–14), or on the Processing menu, click **Generate Functional Simulation Netlist**.



Generating a functional simulation netlist creates a separate database that improves the performance of the simulation significantly.

Figure 3–14. Simulator Tool Dialog Box



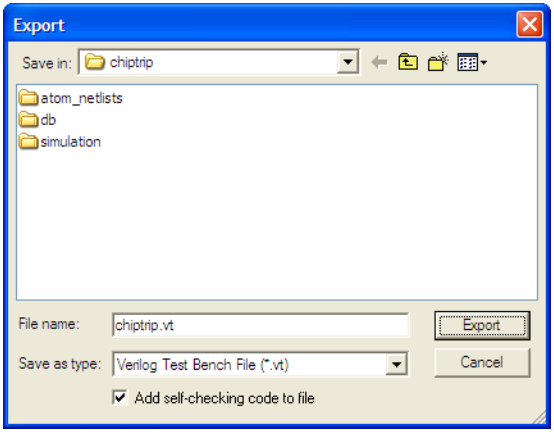
You can view and modify the simulator options on the **Simulator** page of the **Settings** dialog box or in the **Simulator Tool** window. You can set the simulation period and turn **Check outputs** on or off. You can choose to display the simulation outputs in the simulation report or in the Vector Waveform File. To display the simulation results in the simulation input vector waveform file, which is the MAX+PLUS II behavior, turn on **Overwrite simulation input file with simulation results**.

When using either the MAX+PLUS II or Quartus II software, you may have to compile additional behavioral models to perform a simulation with an EDA simulation tool. In the Quartus II software, behavioral models for library of parameterized modules (LPM) functions and Altera-specific megafunctions are available in the **altera_mf** and **220model** library files, respectively. The **220model** and **altera_mf** files can be found in the \<*Quartus II Installation*>**eda\sim_lib** directory.

The Quartus II schematic design files (Block Design File, or **.bdf**) are not compatible with EDA simulation tools. To perform a register transfer level (RTL) functional simulation of a Block Design File using an EDA tool, convert your schematic designs to a VHDL or Verilog HDL design file. Open the schematic design file and on the File menu, click **Create/Update > Create HDL Design File for Current File** to create an HDL design file that corresponds to your Block Design File.

You can export a Vector Waveform File or Simulator Channel File as a Verilog HDL or VHDL test bench file for simulation with an EDA tool. Open your Vector Waveform File or Simulator Channel File and on the File menu, click **Export**. See Figure 3–15. Select **Verilog** or **VHDL Test Bench File (*.vt)** from the **Save as type** list. Turn on **Add self-checking code to file** to add additional self-checking code to the test bench.

Figure 3–15. Export Dialog Box



Place and Route

The Quartus II PowerFit is an incremental fitter that performs place-and-route to fit your design into the targeted device. You can control the Fitter behavior with options in the **Fitter Settings** page of the **Settings** dialog box on the Assignments menu.

High-density device families supported in the Quartus II software, such as the Stratix series, sometimes require significant fitter effort to achieve an optimal fit. The Quartus II software offers several options to reduce the time required to fit a design. You can control the effort the Quartus II Fitter expends to achieve your timing requirements with these options:

- **Optimize timing** performs timing-based placement using the timing requirements you specify for the design. You can use this option by itself or with one or more of the options below.
- **Optimize hold timing** optimizes the hold times within a device to meet timing requirements and assignments you specify. You can select this option only if the Optimize timing option is also chosen.
- **Optimize fast-corner timing** instructs the Fitter, when optimizing your design, to consider fast-corner delays, in addition to slow-corner delays, from the fast-corner timing model (fastest manufactured device, operating in low-temperature and high-voltage conditions). You can select this option only if the **Optimize timing** option is also chosen.

If minimizing compilation time is more important than achieving specific timing results, you can turn these options off.

Another way to decrease the processing time and effort the Fitter expends to fit your design is to select either **Standard Fit** or **Fast Fit** in the **Fitter Effort** box of the **Fitter Settings** page in the **Settings** dialog box on the Assignments menu. The option you select affects the Fitter behavior and your design as described below.

- Select **Standard Fit** for the Fitter to use the highest effort and preserve the performance from previous compilations.
- Select **Fast Fit** for up to 50% faster compilation times, although this may reduce design performance.

You can also select **Auto Fit** to decrease compilation time by directing the Fitter to reduce Fitter effort after meeting your timing requirements. The **Auto Fit** option is available for select devices.



For more information, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

To further reduce compilation times, turn on **Limit to one fitting attempt** in the **Fitter Settings** page in the **Settings** dialog box on the Assignments menu.

If your design is very close to meeting your timing requirements, you can control the seed number used in the fitting algorithm by changing the value in the **Seed** box of the **Fitter Settings** page of the **Settings** dialog box on the Assignments menu. The default seed value is 1. You can specify any non-negative integer value. Changing the value of the seed only repositions the starting location of the Fitter, but does not affect compilation time or the Fitter effort level. However, if your design is difficult to fit optimally or takes a long time to fit, sometimes you can improve results or processing time by changing the seed value.

Timing Analysis

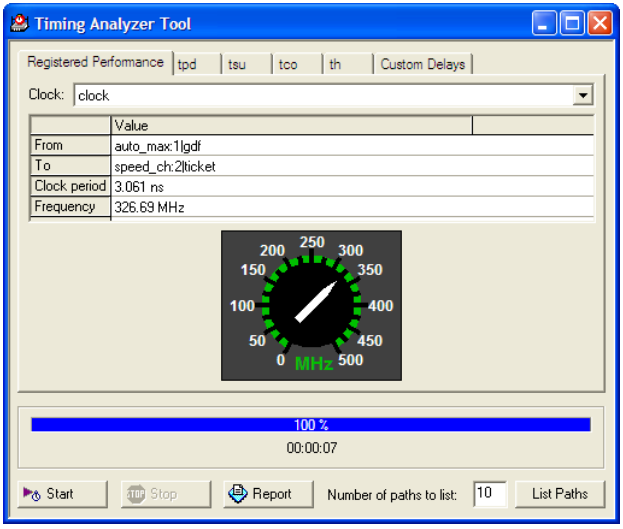
Version 6.1 and later of the Quartus II software supports two native timing analysis tools: TimeQuest Timing Analyzer and the Classic Timing Analyzer. Both timing analysis tools provide more complex clocking schemes than is possible with the MAX_PLUS II Timing Analyzer. The TimeQuest analyzer uses the industry-standard Synopsys Design Constraint (SDC) methodology for constraining designs and reporting results. In general, the TimeQuest Timing Analyzer provides more control in constraining a design as compared to the Classic Timing Analyzer. However, the Classic Timing Analyzer incorporates a basic graphical user interface and the timing analysis flow is similar to the flow in the MAX_PLUS II software. As such, the section that follows provides a more detailed look at timing analysis using the Classic Timing Analyzer.



For more information on choosing between the TimeQuest Timing Analyzer or the Classic Timing Analyzer, refer to the Timing Analysis Section in the *Introduction to Quartus II* manual.

Launch the Classic Timing Analyzer tool on the MAX+PLUS II menu by clicking **Classic Timing Analyzer** or by selecting **Classic Timing Analyzer Tool** on the **Processing** menu. See Figure 3–16. To start the analysis, click **Start** in the Timing Analyzer Tool or on the Processing menu, by pointing to Start, and clicking **Start Timing Analyzer**.

Figure 3–16. Registered Performance Tab of the Timing Analyzer Tool



The Quartus II Classic Timing Analyzer analyzes all clock domains in your design, including paths that cross clock domains. You can ignore paths that cross clock domains by using the following options in the **Timing Requirements & Options** page in the **Settings** dialog box on the Assignments menu:

- Create a **Cut Timing Path** assignment
- Turn on **Cut paths between unrelated clock domains**

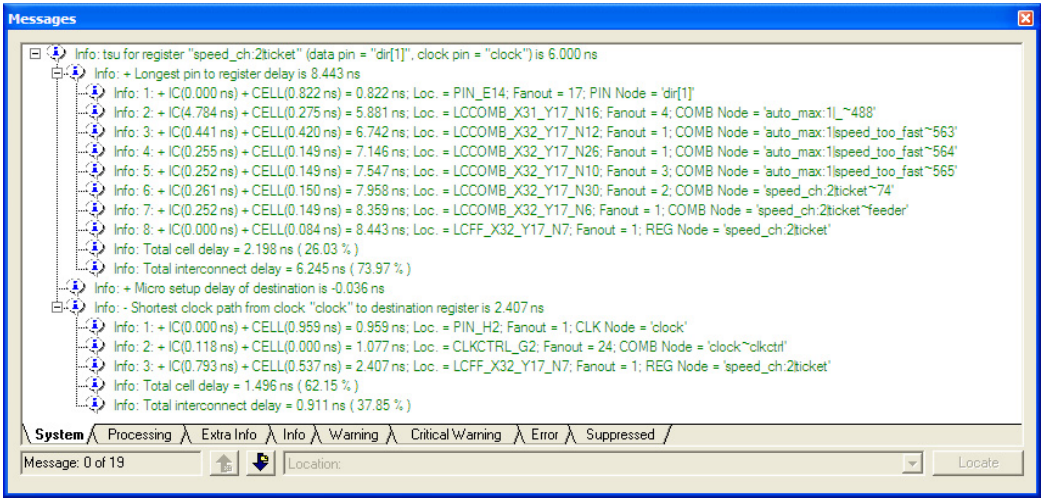
To view the results from the Classic Timing Analyzer Tool, click the **Report** button located at the bottom of the Classic Timing Analyzer dialog box, or to get specific information, click on any of the following tabs at the top of the Classic Timing Analyzer window:

- Registered Performance
- t_{PD}
- t_{SU}
- t_{CO}
- t_{H}
- Custom Delays

The Quartus II Classic Timing Analyzer reports both f_{MAX} and slack. Slack is the margin by which the timing requirement was met or not met. A positive slack value, displayed in black, indicates the margin by which a requirement was met. A negative slack value, displayed in red, indicates the margin by which a requirement was not met.

To analyze a particular path in more detail, select a path in the Classic Timing Analyzer Tool and click **List Paths**. This displays a detailed description of the path in the **System** tab of the **Messages** window (Figure 3–17).

Figure 3–17. Messages Window Displaying Detailed Timing Information



For more information, refer to the *Classic Timing Analyzer* or the *TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Timing Closure Floorplan

The Quartus II Timing Closure Floorplan is similar to the MAX+PLUS II Floorplan Editor but has many improvements to help you more effectively view and debug your design. With its ability to display logic cell usage, routing congestion, critical paths, and LogicLock™ regions, the Timing Closure Floorplan also makes the task of improving your design performance much easier.

To view the Timing Closure Floorplan, on the MAX+PLUS II menu, click **Floorplan Editor** or **Timing Closure Floorplan**.

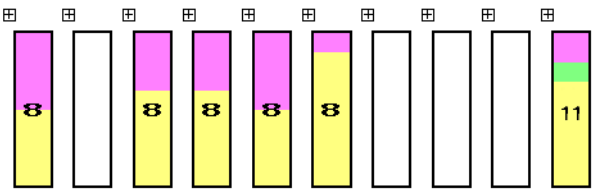
The Timing Closure Floorplan Editor provides Interior Cell views equivalent to the MAX+PLUS II logic array block (LAB) views. In addition to these views, available from the View menu, you also can select from the Interior MegaLABs (where applicable), Interior LABs, and Field views.



The Pin Planner is equivalent to the MAX+PLUS II Device view. The Pin Planner can be launched from the Timing Closure Floorplan Editor by selecting **Package** (Top or Bottom) from the View menu or on the Assignments menu by clicking **Pin Planner**.

The Interior LABs view hides cell details for logic cells, Adaptive Logic Modules (ALM), and macrocells, and shows LAB information (see Figure 3–18). You can display the number of cells used in each LAB on the View menu by clicking **Show Usage Numbers**.

Figure 3–18. Interior LAB View of the Timing Closure Floorplan



The Field view is a color-coded, high-level view of your device resources that hides both cell and LAB details. In the Field view, you can see critical paths and routing congestion in your design.

The View Critical Paths feature shows a percentage of all critical paths in your floorplan. You can enable this feature on the View menu by clicking **Show Critical Paths**. You can control the number of critical paths shown by modifying the settings in the **Critical Paths Settings** dialog box on the View menu.

The View Congestion feature displays routing congestion by coloring and shading logic resources. Darker shading shows greater resource utilization. This feature assists in identifying locations where there is a lack of routing resources.

To show lower level details in any view, right-click on a resource and click **Show Details**.

For more information, refer to the *Timing Closure Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

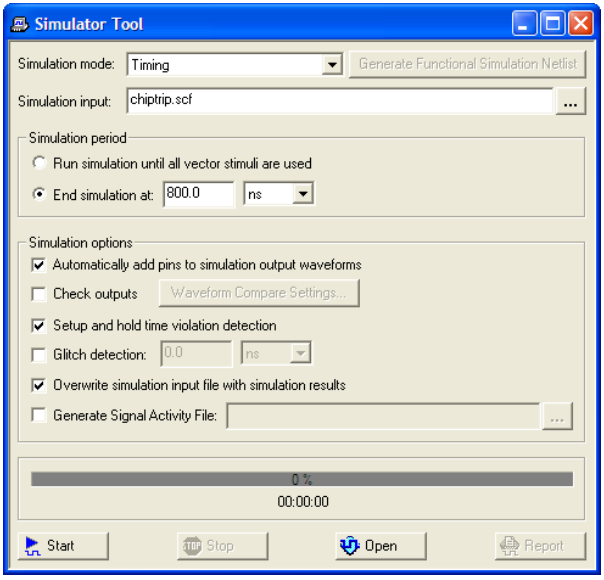
Timing Simulation

Timing simulation is an important part of the verification process. The Quartus II software supports native timing simulation and exports simulation netlists to third-party software for design verification.

Quartus II Simulator Tool

The Quartus II Simulator tool is an easy-to-use integrated solution that uses the compiler database to simulate the logical and timing performance of your design (Figure 3–19). When performing timing simulation, the simulator uses place-and-route timing information.

Figure 3–19. Quartus II Simulator Tool



You can use Vector Table Output Files (**.tbl**), Vector Waveform Files, Vector Files (**.vec**), or an existing Simulator Channel File as the vector stimuli for your simulation.

The simulation options available are similar to the options available in the MAX+PLUS II Simulator. You can control the length of the simulation and the type of checks performed by the Simulator. When the MAX+PLUS II look and feel is selected, the **Overwrite simulation input file with simulation results** option is on by default. If you turn it off, the simulation results are written to the report file. To view the report file, click **Report** in the Simulator Tool window.

EDA Timing Simulation

The Quartus II software also supports timing simulation with other EDA simulation software. Performing timing simulation with other EDA simulation software requires a Quartus II generated timing netlist file in the form of a Verilog Output File (**.vo**) or VHDL Output File (**.vho**), a Standard Delay Format Output File (**.sdo**), and a device-specific atom file (or files), shown in Table 3–3.

Table 3–3. Altera Timing Simulation Library Files

Verilog	VHDL
<*device_family*>_atoms.v	<*device_family*>_atoms_87.vhd
	<*device_family*>_atoms.vhd
	<*device_family*>_components.vhd

Specify your EDA simulation tool by performing the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears.
3. In the **Tool name list**, select your EDA Tool.

You can generate a timing netlist for the selected EDA simulator tool by running a full compile or on the Processing menu, by pointing to Start and clicking **Start EDA Netlist Writer**. The generated netlist and SDF file are placed into the \<*project directory*>**\simulation**\<*EDA simulator tool*> directory. The device-specific atom files are located in the \<*Quartus II Install*>**\eda\sim_lib** directory.

Power Estimation

To develop an appropriate power budget and to design the power supplies, voltage regulators, heat sink, and cooling system, you need an accurate estimate of the power that your design consumes. You can estimate power by using the PowerPlay Early Power Estimation spreadsheet available on the Altera website at **www.altera.com**, or with the PowerPlay Power Analyzer in the Quartus II software.

You can perform early power estimation with the PowerPlay Early Power Estimation spreadsheet by entering device resource and performance information. The Quartus II PowerPlay Analyzer tool performs vector-based power analysis by reading either a Signal Activity File (**.saf**), generated from a Quartus II simulation, or a Value Change Dump File (VCD) generated from a third-party simulation.



For more information about how to use the PowerPlay Power Analyzer tool, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Programming

The Quartus II Programmer has the same functionality as the MAX+PLUS II Programmer, including programming, verifying, examining, and blank checking operations. Additionally, the Quartus II Programmer now supports the erase capability for CPLDs. To improve usability, the Quartus II Programmer displays all programming-related information in one window (Figure 3–20).

Click **Add File** or **Add Device** in the Programmer window to add a file or device, respectively.

Figure 3–20. Programmer Window

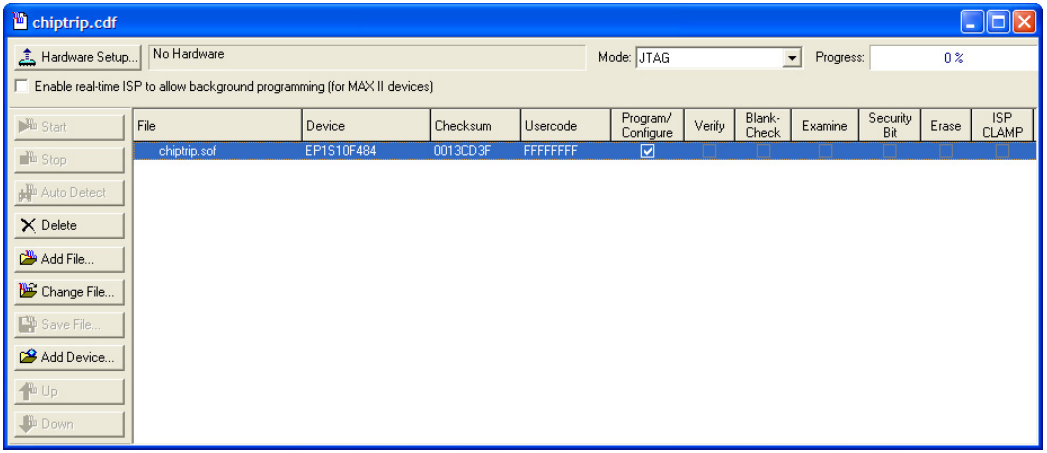




Figure 3–20 shows that the Programmer Window now supports Erase capability.

You can save the programmer settings as a Chain Description File (.**cdf**). The CDF is an ASCII text file that stores device name, device order, and programming file name information.

Conclusion

The Quartus II software is the most comprehensive design environment available for programmable logic designs. Features such as the MAX+PLUS II look and feel help you make the transition from Altera's MAX+PLUS II design software and become more productive with the Quartus II software. The Quartus II software has all the capabilities and features of the MAX+PLUS II software and many more to speed up your design cycle.

Quick Menu Reference

The commands displayed in the MAX+PLUS II Quick Menu and the Quartus II Quick Menu vary based on whichever window is active (Figures 3–21). In the following figure, the Graphic Editor window is active.

Figure 3–21. MAX+PLUS II Quick Menus in MAX+PLUS II and Quartus II Software



Quartus II Command Reference for MAX+PLUS II Users

Table 3–4 lists the commands in the MAX+PLUS II software and gives their equivalent commands in the Quartus II software.

NA means either Not Applicable or Not Available. If a command is not listed, then the command is the same in both tools.

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 1 of 10)

MAX+PLUS II Software	Quartus II Software
MAX+PLUS II Menu	
Hierarchy Display	View menu, Utility Windows, **Project Navigator**
Graphic Editor	Block Editor
Symbol Editor	Block Symbol Editor
Text Editor	Text Editor
Waveform Editor	Waveform Editor
Floorplan Editor	Assignments menu, **Timing Closure Floorplan**
Compiler	Tools menu, **Compiler Tool**
Simulator	Tools menu, **Simulator Tool**
Timing Analyzer	Tools menu, **Timing Analyzer Tool**
Programmer	Tools menu, **Programmer**
Message Processor	View menu, Utility Windows, **Messages**
File Menu	
File menu, Project, **Name** (Ctrl+J)	File menu, **Open Project** (Ctrl+J)
File menu, Project, **Set Project to Current File** (Ctrl+Shift+J)	Project menu, **Set as Top-Level Entity** (Ctrl+Shift+J) or File menu, **New Project Wizard**
File menu, Project, **Save & Check** (Ctrl+K)	Processing menu, Start, **Start Analysis & Synthesis** (Ctrl+K) *or* Processing menu, Start, **Start Analysis & Elaboration**
File menu, Project, **Save & Compile** (Ctrl+L)	Processing menu, **Start Compilation** (Ctrl+L)

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 2 of 10)

MAX+PLUS II Software	Quartus II Software
File menu, Project, **Save & Simulate** (Ctrl+Shift+L)	Processing menu, **Start Simulation** (Ctrl+I)
File menu, Project, **Compile & Simulate** (Ctrl+Shift+K)	Processing menu, **Start Compilation & Simulation** (Ctrl+Shift+K)
File menu, Project, **Archive**	Project menu, **Archive Project**
File menu, Project, <*Recent Projects*>	File menu, <*Recent Projects*>
File menu, **Delete File**	NA
File menu, **Retrieve**	NA
File menu, **Info** (Ctrl+I)	File menu, **File Properties**
File menu, **Create Default Symbol**	File menu, Create/Update, **Create Symbol Files for Current File**
File menu, **Edit Symbol**	(Block Editor) Edit menu, **Edit Selected Symbol**
File menu, **Create Default Include File**	File menu, Create/Update, **Create AHDL Include Files for Current File**
File menu, **Hierarchy Project Top** (Ctrl+T)	Project menu, Hierarchy, **Project Top** (Ctrl+T)
File menu, Hierarchy, **Up** (Ctrl+U)	Project menu, Hierarchy, **Up** (Ctrl+U)
File menu, Hierarchy, **Down** (Ctrl+D)	Project menu, Hierarchy, **Down** (Ctrl+D)
File menu, Hierarchy, **Top**	NA
File menu, Hierarchy, **Project Top** (Ctrl+T)	Project menu, Hierarchy, **Project Top** (Ctrl+T)
File menu, **MegaWizard Plug-In Manager**	Tools menu, **MegaWizard Plug-In Manager**
(Graphic Editor) File menu, **Size**	NA
(Waveform Editor) File menu, **End Time**	(Waveform Editor) Edit menu, **End Time**
(Waveform Editor) File menu, **Compare**	(Waveform Editor) View menu, **Compare to Waveforms in File**
(Waveform Editor) File menu**, Import Vector File**	File menu, **Open** (Ctrl+O)
(Waveform Editor) File menu**, Create Table File**	File menu, **Save As**
(Hierarchy Display) File menu, **Select Hierarchy**	NA
(Hierarchy Display) File menu, **Open Editor**	(Project Navigator) Double-click
(Hierarchy Display) File menu, **Close Editor**	NA
(Hierarchy Display) File menu, **Change File Type**	(Project Navigator) Select file in Files tab and select Properties on right click menu
(Hierarchy Display) File menu, **Print Selected Files**	NA

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 3 of 10)

MAX+PLUS II Software	Quartus II Software
(Programmer) File menu, **Select Programming File**	File menu, **Open**
(Programmer) File menu, **Save Programming Data As**	File menu, **Save**
(Programmer) File menu, **Inputs/Outputs**	NA
(Programmer) File menu, **Convert SRAM Object Files**	File menu, **Convert Programming Files**
(Programmer) File menu, **Archive JTAG Programming Files**	NA
(Programmer) File menu, **Create Jam or SVF File**	File menu, Create/Update, **Create JAM, SVF, or ISC File**
(Message Processor) Select Messages	NA
(Message Processor) Save Messages As	(Messages) Save Messages on right click menu
(Timing Analyzer) Save Analysis As	Processing menu, **Compilation Report** - Save Current Report on right click menu in Timing Analyzer sections
(Simulator) Create Table File	(Waveform Editor) File menu, **Save As**
(Simulator) Execute Command File	NA
(Simulator) Inputs/Outputs	NA
Edit Menu	
(Waveform Editor) Edit menu, **Overwrite**	(Waveform Editor) Edit menu, **Value**
(Waveform Editor) Edit menu, **Insert**	(Waveform Editor) Edit menu, **Insert Waveform Interval**
(Waveform Editor) Edit menu, **Align to Grid** (Ctrl+Y)	NA
(Waveform Editor) Edit menu, **Repeat**	(Waveform Editor) Edit menu, **Repeat Paste**
(Waveform Editor) Edit menu, **Grow or Shrink**	Edit menu, **Grow or Shrink** (Ctrl+Alt+G)
(Text Editor) Edit menu, **Insert Page Break**	(Text Editor) Edit menu, **Insert Page Break**
(Text Editor) Edit menu, **Increase Indent** (F2)	(Text Editor) Edit menu, **Increase Indent**
(Text Editor) Edit menu, **Decrease Indent** (F3)	(Text Editor) Edit menu, **Decrease Indent**
(Graphic Editor) Edit menu, **Toggle Connection Dot** (Double-Click)	(Block Editor) Edit menu, **Toggle Connection Dot**
(Graphic Editor) Edit menu, **Flip Horizontal**	(Block Editor) Edit menu, **Flip Horizontal**
(Graphic Editor) Edit menu, **Flip Vertical**	(Block Editor) Edit menu, **Flip Vertical**
(Graphic Editor) Edit menu, **Rotate**	(Block Editor) Edit menu, **Rotate by Degrees**

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 4 of 10)

MAX+PLUS II Software	Quartus II Software
View Menu	
View menu, **Fit in Window** (Ctrl+W)	View menu, **Fit in Window** (Ctrl+W)
View menu, **Zoom In** (Ctrl+Space)	View menu, **Zoom In** (Ctrl+Space)
View menu, **Zoom Out** (Ctrl+Shift+Space)	View menu, **Zoom Out** (Ctrl+Shift+Space)
View menu, **Normal Size** (Ctrl+1)	NA
View menu, **Maximum Size** (Ctrl+2)	NA
(Hierarchy Display) View menu, **Auto Fit in Window**	NA
(Waveform Editor) View menu, **Time Range**	View menu, **Zoom**
Assign menu, **Device**	Assignments menu, **Device** *or* Assignments menu, **Settings** (Ctrl+Shift+E)
Assign menu, **Pin/Location/Chip**	Assignments menu, **Assignment Editor** - Locations category
Assign menu, **Timing Requirements**	Assignments menu, **Assignment Editor** - Timing category
Assign menu, **Clique**	Assignments menu, **Assignment Editor** - Cliques category
Assign menu, **Logic Options**	Assignments menu, **Assignment Editor** - Logic Options category
Assign menu, **Probe**	NA
Assign menu, **Connected Pins**	Assignments menu, **Assignment Editor** - Simulation category
Assign menu, **Local Routing**	Assignments menu, **Assignment Editor** - Local Routing category
Assign menu, **Global Project Device Options**	Assignments menu, **Device** - Device and Pin Options
Assign menu, **Global Project Parameters**	Assignments menu, **Settings** - Analysis and Synthesis - Default Parameters
Assign menu, **Global Project Timing Requirements**	Assignments menu, **Timing Settings**
Assign menu, **Global Project Logic Synthesis**	Assignments menu, **Settings** - Analysis and Synthesis
Assign menu, **Ignore Project Assignments**	Assignments menu, **Assignment Editor** - disable
Assign menu, **Clear Project Assignments**	Assignments menu, **Remove Assignments**

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 5 of 10)

MAX+PLUS II Software	Quartus II Software
Assign menu, **Back-Annotate Project**	Assignments menu, **Back-Annotate Assignments**
Assign menu, **Convert Obsolete Assignment Format**	NA
Utilities Menu	
 Utilities menu, **Find Text** (Ctrl+F)	Edit menu, **Find** (Ctrl+F)
 Utilities menu, **Find Node in Design File** (Ctrl+B)	 Project menu, Locate, **Locate in Design File**
 Utilities menu, **Find Node in Floorplan**	 Project menu, Locate, **Locate in Timing Closure Floorplan**
Utilities menu, **Find Clique in Floorplan**	NA
Utilities menu, **Find Node Source** (Ctrl+Shift+S)	NA
Utilities menu**, Find Node Destination** (Ctrl+Shift+D)	NA
Utilities menu, **Find Next** (Ctrl+N)	 Edit menu, **Find Next** (F3)
Utilities menu, **Find Previous** (Ctrl+Shift+N)	NA
Utilities menu, **Find Last Edit**	NA
 Utilities menu, **Search and Replace** (Ctrl+R)	 Edit menu, **Replace** (Ctrl+H)
Utilities menu, **Timing Analysis Source** (Ctrl+Alt+S)	NA
Utilities menu, **Timing Analysis Destination** (Ctrl+Alt+D)	NA
Utilities menu, **Timing Analysis Cutoff** (Ctrl+Alt+C)	NA
Utilities menu, **Analyze Timing**	NA
Utilities menu, **Clear All Timing Analysis Tags**	NA
(Text Editor) Utilities menu, **Go To** (Ctrl+G)	 Edit menu, **Go To** (Ctrl+G)
(Text Editor) Utilities menu, **Find Matching Delimiter** (Ctrl+M)	 (Text Editor) Edit, **Find Matching Delimiter** (Ctrl+M)
(Waveform Editor) Utilities menu, **Find Next Transition** (Right Arrow)	(Waveform Editor) View menu, **Next Transition** (Right Arrow)
(Waveform Editor) Utilities menu, **Find Previous Transition** (Left Arrow)	(Waveform Editor) View menu, **Next Transition** (Left Arrow)
Options Menu	
Options menu, **User Libraries**	 Assignments menu, **Settings** (Ctrl+Shift+E) Tools, Options, **Global User LIbraries**

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 6 of 10)

MAX+PLUS II Software	Quartus II Software
Options menu, **Color Palette**	Tools menu, **Options**
Options menu, **License Setup**	Tools menu, **License Setup**
Options menu, **Preferences**	Tools menu, **Options**
(Hierarchy Display) Options menu, **Orientation**	NA
(Hierarchy Display) Options menu, **Compact Display**	NA
(Hierarchy Display) Options menu, **Show All Hierarchy Branches**	(Project Navigator) Expand All on right click menu
(Hierarchy Display) Options menu, **Hide All Hierarchy Branches**	NA
(Editors) Options menu, **Font**	Tools menu, **Options**
(Editors) Options menu, **Text Size**	Tools menu, **Options**
(Graphic Editor) Options menu, **Line Style**	Edit menu, **Line**
 (Graphic Editor) Options menu, **Rubberbanding**	 Tools menu, **Options**
(Graphic Editor) Options menu, **Show Parameters**	 View menu, **Show Parameter Assignments**
(Graphic Editor) Options menu, **Show Probes**	NA
(Graphic Editor) Options menu**, Show Pins/Locations/Chips**	 View menu, **Show Pin and Location Assignments**
(Graphic Editor) Options menu, **Show Clique, Timing & Local Routing Assignments**	NA
(Graphic Editor) Options menu, **Show Logic Options**	NA
 (Graphic Editor) Options menu, **Show All** (Ctrl+Shift+M)	NA
(Graphic Editor) Options menu, **Show Guidelines** (Ctrl+Shift+G)	Tools menu, **Options** - Block/Symbol Editor page
(Graphic Editor) Options menu, **Guideline Spacing**	Tools menu, **Options** - Block/Symbol Editor page
(Symbol Editors) Options menu, **Snap to Grid**	Tools menu, **Options** - Block/Symbol Editor page
(Text Editor) Options menu, **Tab Stops**	Tools menu, **Options** - Text Editor page
(Text Editor) Options menu, **Auto-Indent**	Tools menu, **Options** - Text Editor page
(Text Editor) Options menu, **Syntax Coloring**	NA
(Waveform Editor) Options menu, **Snap to Grid**	 View menu, **Snap to Grid**
(Waveform Editor) Options menu, **Show Grid** (Ctrl+Shift+G)	Tools menu, **Options** - Waveform Editor page
(Waveform Editor) Options menu, **Grid Size**	Edit menu, **Grid Size** - Waveform Editor page

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 7 of 10)

MAX+PLUS II Software	Quartus II Software
(Floorplan Editor) Options menu, **Routing Statistics**	NA
 (Floorplan Editor) Options menu, **Show Node Fan-In**	 View menu, Routing, **Show Fan-In**
 (Floorplan Editor) Options menu, **Show Node Fan-Out**	 View menu, Routing, **Show Fan-Out**
 (Floorplan Editor) Options menu, **Show Path**	 View menu, Routing, **Show Paths between Nodes**
(Floorplan Editor) Options menu, **Show Moved Nodes in Gray**	NA
(Simulator) Options menu, **Breakpoint**	Processing menu, Simulation Debug, **Breakpoints**
(Simulator) Options menu, **Hardware Setup**	NA
(Timing Analyzer) Options menu, **Time Restrictions**	 Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **Auto-Recalculate**	NA
(Timing Analyzer) Options menu, **Cell Width**	NA
(Timing Analyzer) Options menu, **Cut Off I/O Pin Feedback**	 Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **Cut Off Clear & Reset Paths**	 Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **Cut Off Read During Write Paths**	 Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **List Only Longest Path**	NA
(Programmer) Options menu, **Sound**	NA
(Programmer) Options menu, **Programming Options**	Tools menu, **Options** - Programmer page
(Programmer) Options menu, **Select Device**	(Programmer) Edit menu, **Change Device**
(Programmer) Options menu, **Hardware Setup**	(Programmer) Edit menu, **Hardware Setup**
Symbol (Graphic Editor)	
Symbol menu, **Enter Symbol** (Double-Click)	 (Block Editor) Edit menu, **Insert Symbol** (Double-Click)
Symbol menu, **Update Symbol**	 Edit menu, **Update Symbol or Block**
Symbol menu, **Edit Ports/Parameters**	 Edit menu, **Properties**
Element (Symbol Editor)	
Element menu, **Enter Pinstub**	Double-click on edge of symbol

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 8 of 10)

MAX+PLUS II Software	Quartus II Software
Element menu, **Enter Parameters**	NA
Templates (Text Editor)	
 Templates	 (Text Editor) Edit menu, **Insert Template**
Node (Waveform Editor)	
Node menu, **Insert Node** (Double-Click)	Edit menu, **Insert Node or Bus** (Double-Click)
Node menu, **Enter Nodes from SNF**	Edit menu, **Insert Node** - click on Node Finder…
Node menu, **Edit Node**	Double-click on the Node
Node menu, **Enter Group**	Edit menu, **Group**
Node menu, **Ungroup**	Edit menu, **Ungroup**
Node menu, **Sort Names**	 Edit menu, **Sort**
Node menu, **Enter Separator**	NA
Layout (Floorplan Editor)	
Layout menu, **Full Screen**	 View menu, **Full Screen** (Ctrl+Alt+Space)
Layout menu, **Report File Equation Viewer**	 View menu, **Equations**
Layout menu, **Device View** (Double-Click)	 View menu, **Package Top** or  View menu, **Package Bottom**
Layout menu, **LAB View** (Double-Click)	 View menu, **Interior Labs**
 Layout menu, **Current Assignments Floorplan**	 View menu, Assignments, **Show User Assignments**
 Layout menu, **Last Compilation Floorplan**	 View menu, Assignments, **Show Fitter Assignments**
Processing (Compiler)	
Processing menu, **Design Doctor**	 Processing menu, Start, **Start Design Assistant**
Processing menu, **Design Doctor Settings**	 Assignments menu, **Settings** - Design Assistant
Processing menu, **Functional SNF Extractor**	Processing menu, **Generate Functional Simulation Netlist**
Processing menu, **Timing SNF Extractor**	 Processing menu, **Start Analysis & Synthesis**
Processing menu, **Optimize Timing SNF**	NA
Processing menu, **Linked SNF Extractor**	NA

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 9 of 10)

MAX+PLUS II Software	Quartus II Software
Processing menu, **Fitter Settings**	Assignments menu, **Settings** - Fitter Settings
Processing menu, **Report File Settings**	Assignments menu, **Settings**
Processing menu, **Generate AHDL TDO File**	NA
Processing menu, **Smart Recompile**	Assignments menu, **Settings** - Compilation Process
Processing menu, **Total Recompile**	Assignments menu, **Settings** - Compilation Process
Processing menu, **Preserve All Node Name Synonyms**	Assignments menu, **Settings** - Compilation Process
Interfaces (Compiler)	Assignments menu, **EDA Tool Settings**
Initialize (Simulator)	
Initialize menu, **Initialize Nodes/Groups**	NA
Initialize menu, **Initialize Memory**	NA
Initialize menu, **Save Initialization As**	NA
Initialize menu, **Restore Initialization**	NA
Initialize menu, **Reset to Initial SNF Values**	NA
Node (Timing Analyzer)	
Node menu, **Timing Analysis Source** (Ctrl+Alt+S)	NA
Node menu, **Timing Analysis Destination** (Ctrl+Alt+D)	NA
Node menu, **Timing Analysis Cutoff** (Ctrl+Alt+C)	NA
Analysis (Timing Analyzer)	
Analysis menu, **Delay Matrix**	(Timing Analyzer Tool) Delay tab
Analysis menu, **Setup/Hold Matrix**	NA
Analysis menu, **Registered Performance**	(Timing Analyzer Tool) Registered Performance tab
JTAG (Programmer)	
JTAG menu, **Multi-Device JTAG Chain**	(Programmer) Mode: JTAG
JTAG menu, **Multi-Device JTAG Chain Setup**	(Programmer) Window
JTAG menu, **Save JCF**	File menu, **Save**
JTAG menu, **Restore JCF**	File menu, **Open**
JTAG menu, **Initiate Configuration from Configuration Device**	Tools menu, **Options** - Programmer page

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 10 of 10)

MAX+PLUS II Software	Quartus II Software
FLEX (Programmer)	
FLEX menu, **Multi-Device FLEX Chain**	(Programmer) Mode: Passive Serial
FLEX menu, **Multi-Device FLEX Chain Setup**	(Programmer) Window
FLEX menu, **Save FCF**	File menu, **Save**
FLEX menu, **Restore FCF**	File menu, **Open**

Referenced Documents

This chapter references the following documents:

- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Command Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Engineering Change Management with the Chip Planner* chapter in volume 3 of the *Quartus II Handbook*
- *Introduction to Quartus II* manual
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 3–5 show the revision history of this chapter.

Table 3–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents".	Updated for the Quartus II 7.2 software release.
May 2007 v7.1.0	• Added support for Arria GX in Table 3–1. • Added "Referenced Documents" section.	Minor updates to support Altera's newest device, Arria GX.
March 2007 v7.0.0	Consolidated the device support table (Table 1-3) to show support for Stratix series and Cyclone series devices.	—
November 2006 v6.1.0	Added document revision history to chapter.	—
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.	—
December 2005 v5.1.1	Minor typographic and formatting updates.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
May 2005 v5.0.0	Chapter 2 was formerly Chapter 1 in version 4.2.	—
Dec. 2004 v2.1.0	Updated for Quartus II software version 4.2. • Chapter 1 was formerly Chapter 2. • General formatting, editing updates, and figure updates. • FLEX® 600 device support added. • Assignment Editor, Timing Assignments, and Synthesis updated. • APEX II support for balanced optimization technique removed, MAX II support added. • Minor updates to Place and Route. • Tcl commands no longer supported for the Quartus II Simulator Tool. • Excel-based power calculator replaced by PowerPlay Early Power Estimation spreadsheet. • Added support for erase capability for CPLDs.	—
June 2004 v2.0	• Updates to tables, figures. • New functionality for Quartus II software 4.1.	—
Feb. 2004 v1.0	Initial release.	—

4. Quartus II Support for HardCopy Series Devices

QII51004-7.2.0

Introduction

This chapter includes Quartus® II Support for HardCopy® II and HardCopy Stratix® devices. This chapter is divided into the following sections:

- "HardCopy II Device Support" on page 4–1
- "HardCopy Stratix Device Support" on page 4–34

HardCopy II Device Support

Altera® HardCopy II devices feature 1.2-V, 90 nm process technology, and provide a structured ASIC alternative to increasingly expensive multi-million gate ASIC designs. The HardCopy II design methodology offers a fast time-to-market schedule, providing ASIC designers with a solution to long ASIC development cycles. Using the Quartus II software, you can leverage a Stratix II FPGA as a prototype and seamlessly migrate your design to a HardCopy II device for production.

This document discusses the following topics:

- "HardCopy II Development Flow" on page 4–3
- "HardCopy II Device Resource Guide" on page 4–8
- "HardCopy II Recommended Settings in the Quartus II Software" on page 4–12
- "HardCopy II Utilities Menu" on page 4–25



For more information about HardCopy II, HardCopy Stratix, and HardCopy APEX™ devices, refer to the respective device data sheets in the *HardCopy Series Handbook*.

HardCopy II Design Benefits

Designing with HardCopy II structured ASICs offers substantial benefits over other structured ASIC offerings:

- Prototyping using a Stratix II FPGA for functional verification and system development reduces total project development time
- Seamless migration from a Stratix II FPGA prototype to a HardCopy II device reduces time to market and risk
- Unified design methodology for Stratix II FPGA design and HardCopy II design reduces the need for ASIC development software

- Low up-front development cost of HardCopy II devices reduces the financial risk to your project

Quartus II Features for HardCopy II Planning

With the Quartus II software you can design a HardCopy II device using a Stratix II device as a prototype. The Quartus II software contains the following expanded features for HardCopy II device planning:

- **HardCopy II Companion Device Assignment**—Identifies compatible HardCopy II devices for migration with the Stratix II device currently selected.

  This feature constrains the pins of your Stratix II FPGA prototype making it compatible with your HardCopy II device. It also constrains the correct resources available for the HardCopy II device making sure that your Stratix II FPGA design does not become incompatible. In addition, you are still required to compile the design targeting the HardCopy II device to ensure that the design fits, routes, and meets timing.

- **HardCopy II Utilities**—The HardCopy II Utilities functions create or overwrites HardCopy II companion revisions, change revisions to use, and compare revisions for equivalency.
- **HardCopy II Advisor**—The HardCopy II Advisor helps you follow the necessary steps to successfully submit a HardCopy II design to Altera's HardCopy Design Center.

  The HardCopy II Advisor is similar to the Resource Optimization Advisor and Timing Optimization Advisor. The HardCopy II Advisor provides guidelines you can follow during development, reporting the tasks completed as well as the tasks that remain to be completed during development.

- **HardCopy II Floorplan**—The Quartus II software can show a preliminary floorplan view of your HardCopy II design's Fitter placement results.
- **HardCopy II Design Archiving**—The Quartus II software archives the HardCopy II design project's files needed to handoff the design to the HardCopy Design Center.

  This feature is similar to the Quartus II software HardCopy Files Wizard used for HardCopy Stratix and HardCopy APEX families.

- **HardCopy II Device Preliminary Timing**—The Quartus II software performs a timing analysis of HardCopy II devices based on preliminary timing models and Fitter placements. Final timing results for HardCopy II devices are provided by the HardCopy Design Center.
- **HardCopy II Handoff Report**-—The Quartus II software generates a handoff report containing information about the HardCopy II design used by the HardCopy Design Center in the design review process.
- **Formal Verification**—Cadence Encounter Conformal software can now perform formal verification between the source RTL design files and post-compile gate level netlist from a HardCopy II design.

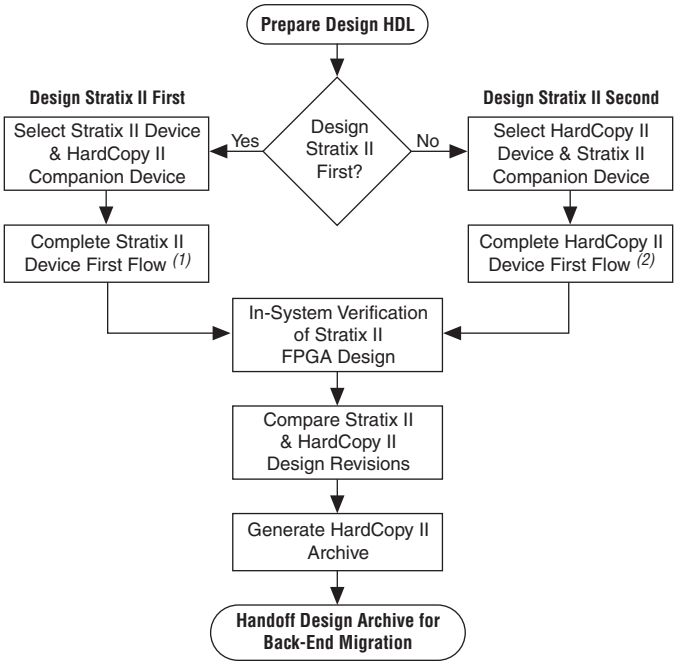
HardCopy II Development Flow

In the Quartus II software, you have two methods for designing your Stratix II FPGA and HardCopy II companion device together in one Quartus II project.

- Design the HardCopy II device first, and create the Stratix II FPGA companion device second and build your prototype for in-system verification
- Design the Stratix II FPGA first and create a HardCopy II companion device second

Both of these flows are illustrated at a high level in Figure 4–1. The added features in the HardCopy II Utilities menu assist you in completing your HardCopy II design for submission to Altera's HardCopy Design Center for back-end implementation.

Figure 4–1. HardCopy II Flow in Quartus II Software



Notes for Figure 4–1:
(1) Refer to Figure 4–2 on page 4–5 for an expanded description of this process.
(2) Refer to Figure 4–3 on page 4–7 for an expanded description of this process.

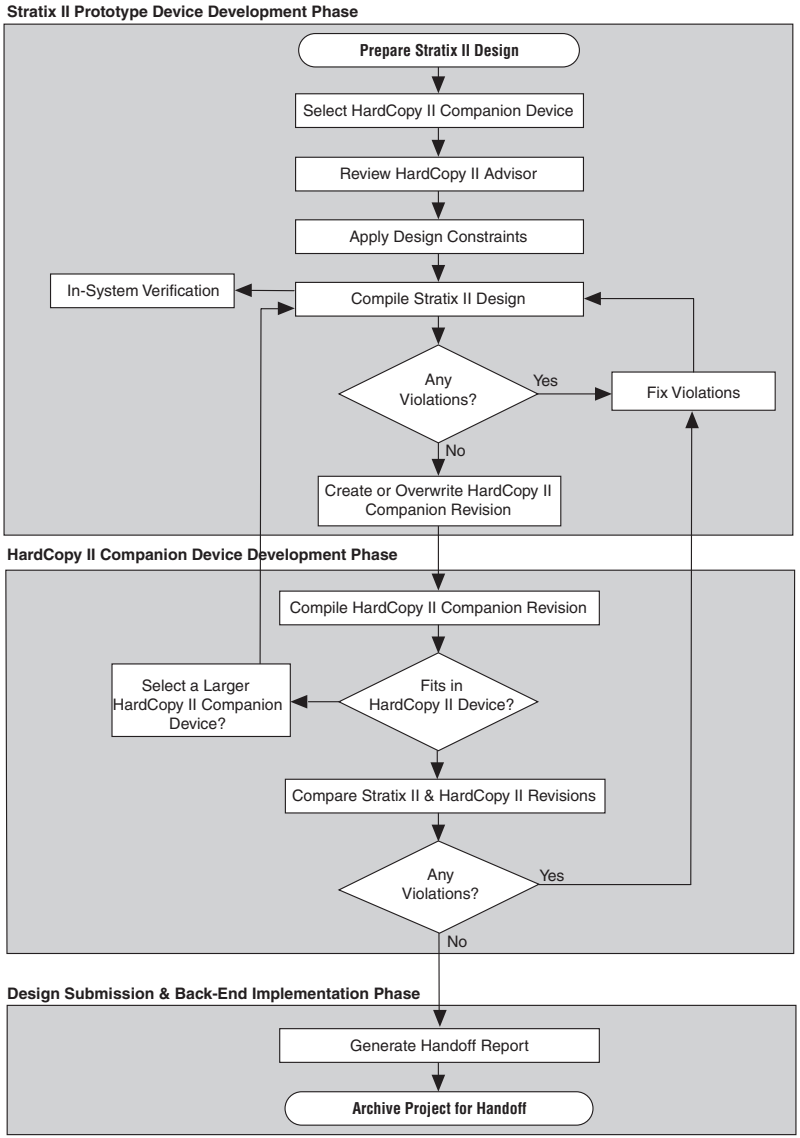
Designing the Stratix II FPGA First

The HardCopy II development flow beginning with the Stratix II FPGA prototype is very similar to a traditional Stratix II FPGA design flow, but requires a few additional tasks be performed to migrate the design to the HardCopy II companion device. To design your HardCopy II device using the Stratix II FPGA as a prototype, complete the following tasks:

- Specify a HardCopy II device for migration
- Compile the Stratix II FPGA design
- Create and compile the HardCopy II companion revision
- Compare the HardCopy II companion revision compilation to the Stratix II device compilation

Figure 4–2 provides an overview highlighting the development process for designing with a Stratix II FPGA first and creating a HardCopy II companion device second.

Figure 4–2. Designing Stratix II Device First Flow



Prototype your HardCopy II design by selecting and then compiling a Stratix II device in the Quartus II software.

After you compile the Stratix II design successfully, you can view the HardCopy II Device Resource Guide in the Quartus II software Fitter report to evaluate which HardCopy II devices meet your design's resource requirements. When you are satisfied with the compilation results and the choice of Stratix II and HardCopy II devices, on the Assignments menu, click **Settings**. In the **Category** list, select **Device**. In the **Device** page, select a HardCopy II companion device.

After you select your HardCopy II companion device, do the following:

- Review the HardCopy II Advisor for required and recommended tasks to perform
- Enable Design Assistant to run during compilation
- Add timing and location assignments
- Compile your Stratix II design
- Create your HardCopy II companion revision
- Compile your design for the HardCopy II companion device
- Use the HardCopy II Utilities to compare the HardCopy II companion device compilation with the Stratix II FPGA revision
- Generate a HardCopy II Handoff Report using the HardCopy II Utilities
- Generate a HardCopy II Handoff Archive using the HardCopy II Utilities
- Arrange for submission of your HardCopy II handoff archive to Altera's HardCopy Design Center for back-end implementation



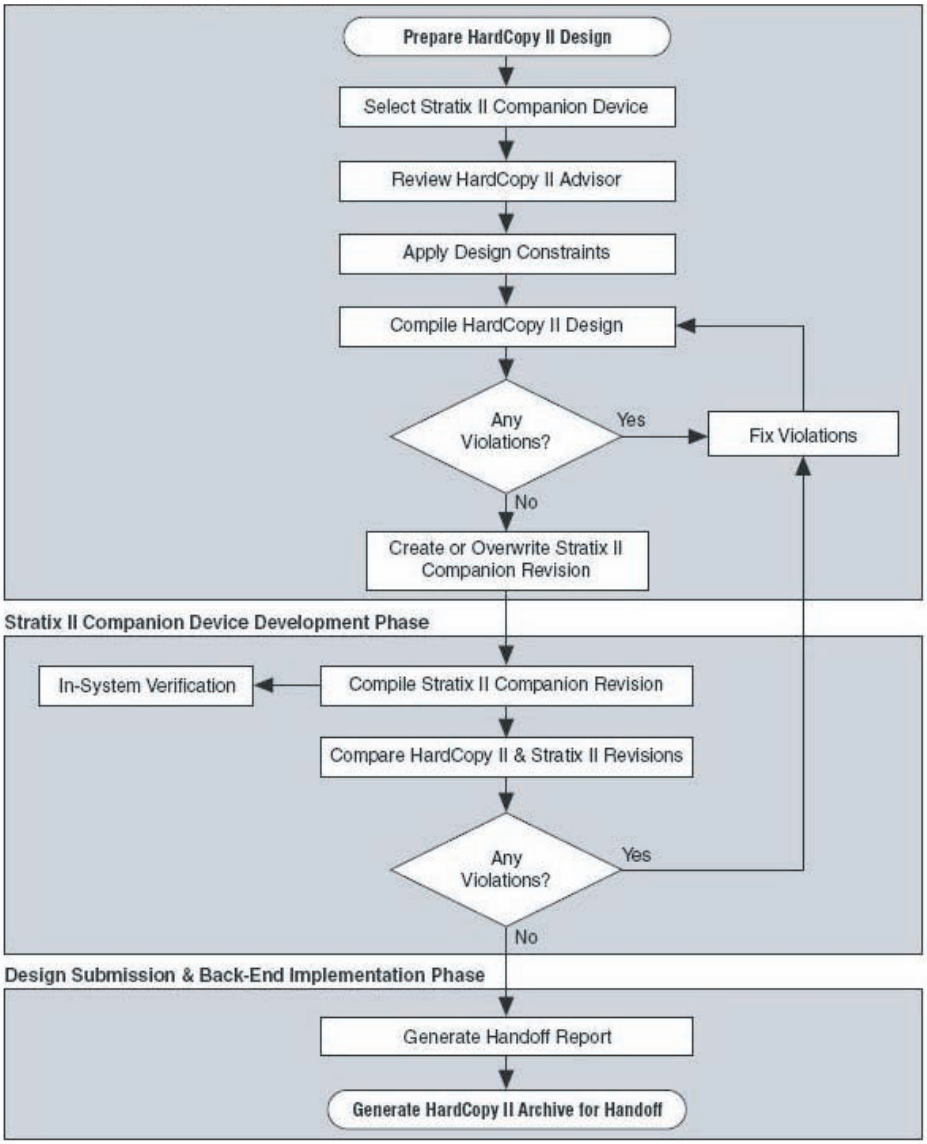
For more information about the overall design flow using the Quartus II software, refer to the *Introduction to Quartus II* manual on the Altera website at **www.altera.com**.

Designing the HardCopy II Device First

The HardCopy II family presents a new option in designing unavailable in previous HardCopy families. You can design your HardCopy II device first and create your Stratix II FPGA prototype second in the Quartus II software. This allows you to see your potential maximum performance in the HardCopy II device immediately during development, and you can create a slower performing FPGA prototype of the design for in-system verification. This design process is similar to the traditional HardCopy II design flow where you build the FPGA first, but instead, you merely change the starting device family. The remaining tasks to complete your design for both Stratix II and HardCopy II devices roughly follow the

same process (Figure 4–3). The HardCopy II Advisor adjusts its list of tasks based on which device family you start with, Stratix II or HardCopy II, to help you complete the process seamlessly.

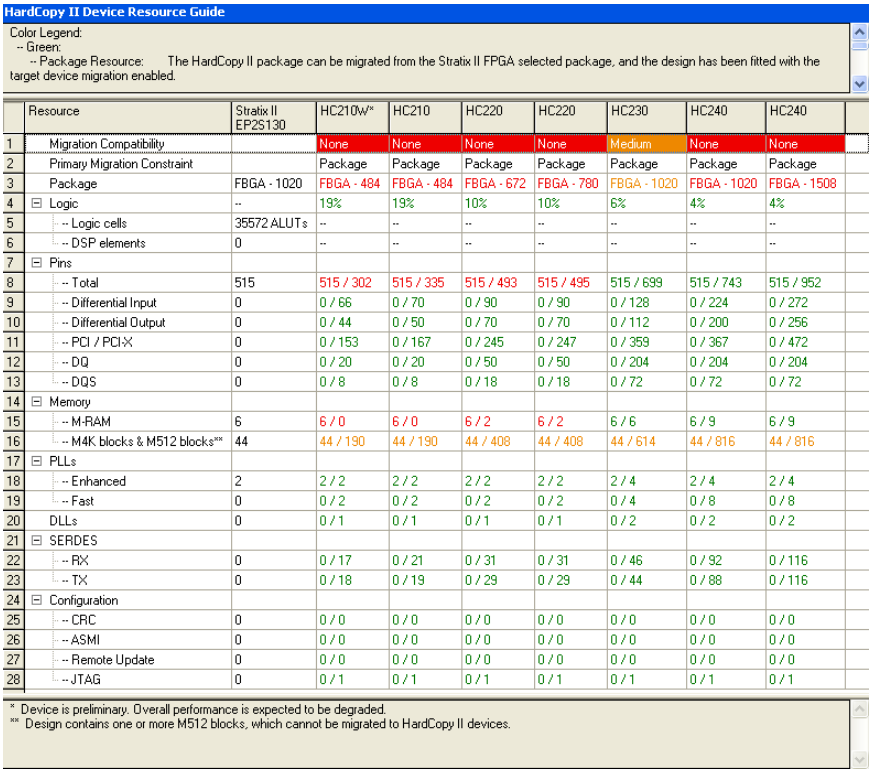
Figure 4–3. Designing HardCopy II Device First Flow



HardCopy II Device Resource Guide

The HardCopy II Device Resource Guide compares the resources required to successfully compile a design with the resources available in the various HardCopy II devices. The report rates each HardCopy II device and each device resource for how well it fits the design. The Quartus II software generates the HardCopy II Device Resource Guide for all designs successfully compiled for Stratix II devices. This guide is found in the Fitter folder of the Compilation Report. Figure 4–4 shows an example of the HardCopy II Device Resource Guide. Refer to Table 4–1 for an explanation of the color codes in Figure 4–4.

Figure 4–4. HardCopy II Device Resource Guide



HardCopy II Device Resource Guide

Color Legend:
-- Green:
-- Package Resource: The HardCopy II package can be migrated from the Stratix II FPGA selected package, and the design has been fitted with the target device migration enabled.

	Resource	Stratix II EP2S130	HC210W*	HC210	HC220	HC220	HC230	HC240	HC240
1	Migration Compatibility		None	None	None	None	Medium	None	None
2	Primary Migration Constraint		Package	Package	Package	Package	Package	Package	Package
3	Package	FBGA - 1020	FBGA - 484	FBGA - 484	FBGA - 672	FBGA - 780	FBGA - 1020	FBGA - 1020	FBGA - 1508
4	⊟ Logic	--	19%	19%	10%	10%	6%	4%	4%
5	-- Logic cells	35572 ALUTs	--	--	--	--	--	--	--
6	-- DSP elements	0	--	--	--	--	--	--	--
7	⊟ Pins								
8	-- Total	515	515 / 302	515 / 335	515 / 493	515 / 495	515 / 699	515 / 743	515 / 952
9	-- Differential Input	0	0 / 66	0 / 70	0 / 90	0 / 90	0 / 128	0 / 224	0 / 272
10	-- Differential Output	0	0 / 44	0 / 50	0 / 70	0 / 70	0 / 112	0 / 200	0 / 256
11	-- PCI / PCI-X	0	0 / 153	0 / 167	0 / 245	0 / 247	0 / 359	0 / 367	0 / 472
12	-- DQ	0	0 / 20	0 / 20	0 / 50	0 / 50	0 / 204	0 / 204	0 / 204
13	-- DQS	0	0 / 8	0 / 8	0 / 18	0 / 18	0 / 72	0 / 72	0 / 72
14	⊟ Memory								
15	-- M-RAM	6	6 / 0	6 / 0	6 / 2	6 / 2	6 / 6	6 / 9	6 / 9
16	-- M4K blocks & M512 blocks**	44	44 / 190	44 / 190	44 / 408	44 / 408	44 / 614	44 / 816	44 / 816
17	⊟ PLLs								
18	-- Enhanced	2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 4	2 / 4	2 / 4
19	-- Fast	0	0 / 2	0 / 2	0 / 2	0 / 2	0 / 4	0 / 8	0 / 8
20	DLLs	0	0 / 1	0 / 1	0 / 1	0 / 1	0 / 2	0 / 2	0 / 2
21	⊟ SERDES								
22	-- RX	0	0 / 17	0 / 21	0 / 31	0 / 31	0 / 46	0 / 92	0 / 116
23	-- TX	0	0 / 18	0 / 19	0 / 29	0 / 29	0 / 44	0 / 88	0 / 116
24	⊟ Configuration								
25	-- CRC	0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
26	-- ASMI	0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
27	-- Remote Update	0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
28	-- JTAG	0	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1

* Device is preliminary. Overall performance is expected to be degraded.
** Design contains one or more M512 blocks, which cannot be migrated to HardCopy II devices.

Use this report to determine which HardCopy II device is a potential candidate for migration of your Stratix II design. The HardCopy II device package must be compatible with the Stratix II device package. A logic

resource usage greater than 100% or a ratio greater than 1/1 in any category indicates that the design does not fit in that particular HardCopy II device.

Table 4–1. HardCopy II Device Resource Guide Color Legend

Color	Package Resource *(1)*	Device Resources
Green (High)	The design can migrate to the Hardcopy II package and the design has been fitted with target device migration enabled in the **HardCopy II Companion Device** dialog box.	The resource quantity is within the range of the HardCopy II device and the design can likely migrate if all other resources also fit. You are still required to compile the HardCopy II revision to make sure the design is able to route and migrate all other resources.
Orange (Medium)	The design can migrate to the Hardcopy II package. However, the design has not been fitted with target device migration enabled in the **HardCopy II Companion Device** dialog box.	The resource quantity is within the range of the HardCopy II device. However, the resource is at risk of exceeding the range for the HardCopy II package. If your target HardCopy II device falls in this category, compile your design targeting the HardCopy II device as soon as possible to check if the design fits and is able to route and migrate all other resources. You may need to migrate to a larger device.
Red (None)	The design cannot migrate to the Hardcopy II package.	The resource quantity exceeds the range of the HardCopy II device. The design cannot migrate to this HardCopy II device.

Note to Table 4–1:

(1) The package resource is constrained by the Stratix II FPGA for which the design was compiled. Only vertical migration devices within the same package are able to migrate to HardCopy II devices.

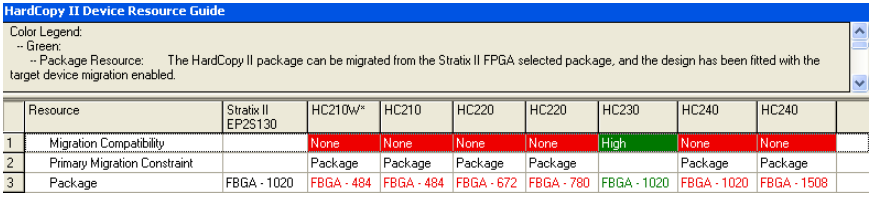
The HardCopy II architecture consists of an array of fine-grained HCells, which are used to build logic equivalent to Stratix II adaptive logic modules (ALMs) and digital signal processing (DSP) blocks. The DSP blocks in HardCopy II devices match the functionality of the Stratix II DSP blocks, though timing of these blocks is different than the FPGA DSP blocks because they are constructed of HCell Macros. The M4K and M-RAM memory blocks in HardCopy II devices are equivalent to the Stratix II memory blocks. Preliminary timing reports of the HardCopy II device are available in the Quartus II software. Final timing results of the HardCopy II device are provided by the HardCopy Design Center after back-end migration is complete.

For more information about the HardCopy II device resources, refer to the *Introduction to HardCopy II Devices* and the *Description, Architecture and Features* chapters in the *HardCopy II Device Family Data Sheet* in the *HardCopy Series Handbook*.

The report example in Figure 4–4 shows the resource comparisons for a design compiled for a Stratix II EP2S130F1020 device. Based on the report, the HC230F1020 device in the 1,020-pin FineLine BGA® package is an appropriate HardCopy II device to migrate to. If the HC230F1020 device is not specified as a migration target during the compilation, its package and migration compatibility is rated orange, or Medium. The migration compatibilities of the other HardCopy II devices are rated red, or None, because the package types are incompatible with the Stratix II device. The 1,020-pin FBGA HC240 device is rated red because it is only compatible with the Stratix II EP2S180F1020 device.

Figure 4–5 shows the report after the (unchanged) design was recompiled with the HardCopy II HC230F1020 device specified as a migration target. Now the HC230F1020 device package and migration compatibility is rated green, or High.

Figure 4–5. HardCopy II Device Resource Guide with Target Migration Enabled



HardCopy II Device Resource Guide

Color Legend:
-- Green:
-- Package Resource: The HardCopy II package can be migrated from the Stratix II FPGA selected package, and the design has been fitted with the target device migration enabled.

	Resource	Stratix II EP2S130	HC210W*	HC210	HC220	HC220	HC230	HC240	HC240
1	Migration Compatibility		None	None	None	None	High	None	None
2	Primary Migration Constraint		Package	Package	Package	Package		Package	Package
3	Package	FBGA - 1020	FBGA - 484	FBGA - 484	FBGA - 672	FBGA - 780	FBGA - 1020	FBGA - 1020	FBGA - 1508

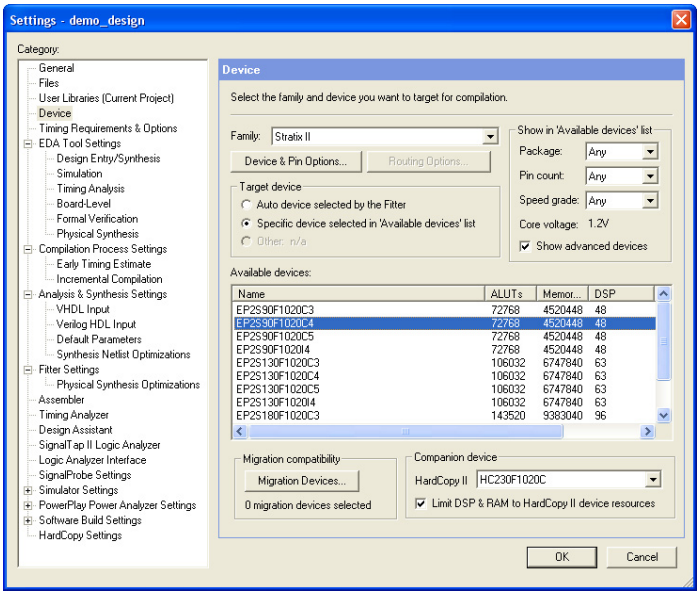
HardCopy II Companion Device Selection

In the Quartus II software, you can select a HardCopy II companion device to help structure your design for migration from a Stratix II device to a HardCopy II device. To make your HardCopy II companion device selection, on the Assignments menu, click **Settings**. In the **Settings** dialog box in the **Category** list, select **Device** (Figure 4–6) and select your companion device from the **Available devices** list.

Selecting a HardCopy II Companion device to go with your Stratix II prototype constrains the memory blocks, DSP blocks, and pin assignments, so that your Stratix II and HardCopy II devices are migration-compatible. Pin assignments are constrained in the Stratix II design revision so that the HardCopy II device selected is

pin-compatible. The Quartus II software also constrains the Stratix II design revision so it does not use M512 memory blocks or exceed the number of M-RAM blocks in the HardCopy II companion device.

Figure 4–6. Quartus II Settings Dialog Box



You can also specify your HardCopy II companion device using the following tool command language (Tcl) command:

```
set_global_assignment -name\
DEVICE_TECHNOLOGY_MIGRATION_LIST <HardCopy II Device Part Number>
```

For example, to select the HC230F1020 device as your HardCopy II companion device for the EP2S130F1020C4 Stratix II FPGA, the Tcl command is:

```
set_global_assignment -name\
DEVICE_TECHNOLOGY_MIGRATION_LIST HC230F1020C
```

HardCopy II Recommended Settings in the Quartus II Software

The HardCopy II development flow involves additional planning and preparation in the Quartus II software compared to a standard FPGA design. This is because you are developing your design to be implemented in two devices: a prototype of your design in a Stratix II prototype FPGA, and a companion revision in a HardCopy II device for production. You need additional settings and constraints to make the Stratix II design compatible with the HardCopy II device and, in some cases, you must remove certain settings in the design. This section explains the additional settings and constraints necessary for your design to be successful in both Stratix II FPGA and HardCopy II structured ASIC devices.

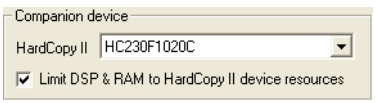
Limit DSP and RAM to HardCopy II Device Resources

On the Assignments menu, click **Settings** to view the **Settings** dialog box. In the **Category** list, select **Device**. In the **Family** list, select **Stratix II**. Under **Companion device**, **Limit DSP & RAM to HardCopy II device resources** is turned on by default (Figure 4–7). This maintains compatibility between the Stratix II and HardCopy II devices by ensuring your design does not use resources in the Stratix II device that are not available in the selected HardCopy II device.



If you require additional memory blocks or DSP blocks for debugging purposes using SignalTap® II, you can temporarily turn this setting off to compile and verify your design in your test environment. However, your final Stratix II and HardCopy II designs submitted to Altera for back-end migration must be compiled with this setting turned on.

Figure 4–7. Limit DSP & RAM to HardCopy II Device Resources Check Box



Enable Design Assistant to Run During Compile

You must use the Quartus II Design Assistant to check all HardCopy series designs for design rule violations before submitting the designs to the Altera HardCopy Design Center. Additionally, you must fix all critical and high-level errors.



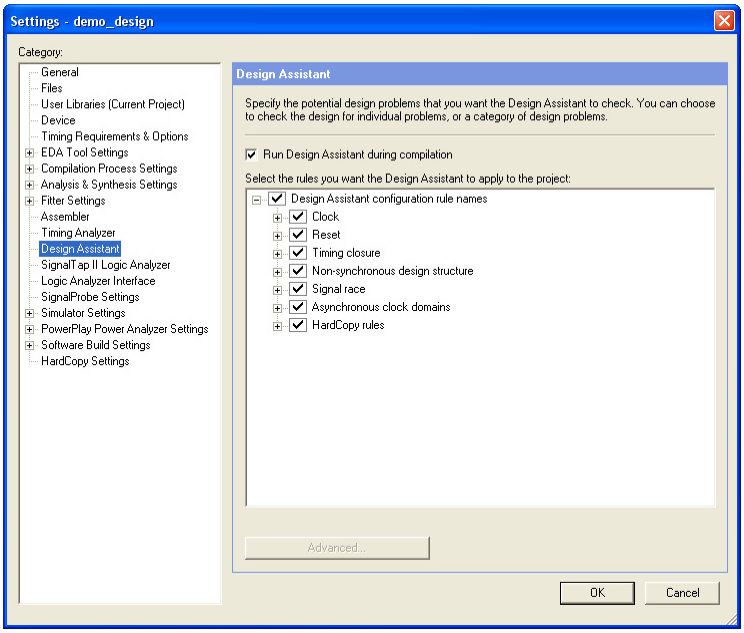
Altera recommends turning on the Design Assistant to run automatically during each compile, so that during development, you can see the violations you must fix.

For more information about the Design Assistant and the rules it uses, refer to the *Design Guidelines for HardCopy Series Devices* chapter of the *HardCopy Series Handbook*.

To enable the Design Assistant to run during compilation, on the Assignment menu, click **Settings**. In the **Category** list, select **Design Assistant** and turn on **Run Design Assistant during compilation** (Figure 4–8) or by entering the following Tcl command in the Tcl Console:

```
set_global_assignment -name ENABLE_DRC_SETTINGS ON
```

Figure 4–8. Enabling Design Assistant



Timing Settings

Beginning in Quartus II Software version 7.1, TimeQuest is the recommended timing analysis tool for all designs. Classic Timing Analyzer is no longer supported and the HardCopy Design Center will not accept any designs which use Classic Timing Analyzer for timing closure.

If you are still using the Classic Timing Analyzer, Altera strongly recommends that you switch to TimeQuest.



For more information on how to switch to TimeQuest, refer to the *Switching to the TimeQuest Timing Analyzer* chapter of the *Quartus II Handbook*, volume 3, on the Altera website at **www.altera.com**.

When you specify the TimeQuest analyzer as the timing analysis tool, the TimeQuest analyzer guides the Fitter and analyzes timing results after compilation.

TimeQuest

The TimeQuest Timing Analyzer is a powerful ASIC-style timing analysis tool that validates timing in your design by using an industry-standard constraint, analysis, and reporting methodology. You can use the TimeQuest Timing Analyzer's GUI or command-line interface to constrain, analyze, and report results for all timing paths in your design.

Before running the TimeQuest Timing Analyzer, you must specify initial timing constraints that describe the clock characteristics, timing exceptions, and signal transition arrival and required times. You can specify timing constraints in the Synopsys Design Constraints (SDC) file format using the GUI or command-line interface. The Quartus II Fitter optimizes the placement of logic to meet your constraints.

During timing analysis, the TimeQuest Timing Analyzer analyzes the timing paths in the design, calculates the propagation delay along each path, checks for timing constraint violations, and reports timing results as slack in the Report pane and in the Console pane. If the TimeQuest Timing Analyzer reports any timing violations, you can customize the reporting to view precise timing information about specific paths, and then constrain those paths to correct the violations. When your design is free of timing violations, you can be confident that the logic will operate as intended in the target device.

The TimeQuest Timing Analyzer is a complete static timing analysis tool that you can use as a sign-off tool for Altera FPGAs and structured ASICs.

Setting Up the TimeQuest Timing Analyzer

If you want use TimeQuest for timing analysis, from the Assignments tab in the Quartus II software, click on **Timing Analysis Settings**, and in the pop-up window, click the **Use TimeQuest Timing Analyzer during compilation** tab.

Use the following Tcl command to use TimeQuest as your timing analysis engine:

```
set_global_assignment -name \
USE_TIMEQUEST_TIMING_ANALYZER ON
```

You can launch the TimeQuest analyzer in one of the following modes:

- Directly from the Quartus II software
- Stand-alone mode
- Command-line mode

In order to perform a thorough Static Timing Analysis, you would need to specify all the timing requirements. The most important timing requirements are clocks and generated clocks, input and output delays, false paths and multi-cycle paths, minimum and maximum delays.

In TimeQuest, clock latency, and recovery and removal analysis are enabled by default.



For more information about TimeQuest, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* on the Altera website at **www.altera.com**.

Constraints for Clock Effect Characteristics

The `create_clock`, `create_generated_clock` commands create ideal clocks and do not account for board effects. In order to account for clock effect characteristics, you can use the following commands:

- `set_clock_latency`
- `set_clock_uncertainty`



For more information about how to use these commands, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Beginning in Quartus II version 7.1, you can use the new command `derive_clock_uncertainty` to automatically derive the clock uncertainties. This command is useful when you are not sure what the clock uncertainties might be. The calculated clock uncertainty values are based on I/O buffer, static phase errors (SPE) and jitter in the PLL's, clock networks, and core noises.

The `derive_clock_uncertainty` command applies inter-clock, intra-clock, and I/O interface uncertainties. This command automatically calculates and applies setup and hold clock uncertainties for each clock-to-clock transfer found in your design.

In order to get I/O interface uncertainty, you must create a virtual clock, then assign delays to the input/output ports by using the `set_input_delay` and `set_output_delay` commands for that virtual clock.



These uncertainties are applied in addition to those you specified using the `set_clock_uncertainty` command. However, if a clock uncertainty assignment for a source and destination pair was already defined, the new one will be ignored. In this case, you can use either the `-overwrite` command to overwrite the previous clock uncertainty command or manually remove them by using the `remove_clock_uncertainty` command.

The syntax for the derive_clock_uncertainty is as follows:

```
derive_clock_uncertainty [-h | -help] [-long_help]
[-dtw] [-overwrite]
```

where the arguments are listed in Table 4–2:

Table 4–2. Arguments for derive_clock_uncertainty

Option	Description
-h | -help	Short help
-long_help	Long help with examples and possible return values
-dtw	Creates PLLJ_PLLSPE_INFO.**txt** file
-overwrite	Overwrites previously performed clock uncertainty assignments

When the `dtw` option is used, a PLLJ_PLLSPE_INFO.**txt** file is generated. This file lists the name of the PLLs, as well as their jitter and SPE values in the design. This text file can be used by HCII_DTW_CU_Calculator. When this option is used, clock uncertainties are not calculated.



For more information on the `derive_clock_uncertainty` command, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Altera strongly recommends that you use the `derive_clock_uncertainty` command in the HardCopy II revision. The HardCopy Design Center will not be accepting designs that do not have clock uncertainty constraint by either using the `derive_clock_uncertainty` command or the HardCopy II Clock Uncertainty Calculator, and then using the `set_clock_uncertainty` command.



For more information on how to use the HardCopy II Clock Uncertainty Calculator, refer to the *HardCopy II Clock Uncertainty User Guide* available on the Altera website at **www.altera.com**.

Quartus II Software Features Supported for HardCopy II Designs

The Quartus II software supports optimization features for HardCopy II prototype development, including:

- Physical Synthesis Optimization
- LogicLock Regions
- PowerPlay Power Analyzer
- Incremental Compilation (Synthesis and Fitter)
- Maximum Fan-Out Assignments

Physical Synthesis Optimization

To enable Physical Synthesis Optimizations for the Stratix II FPGA revision of the design, on the Assignments menu, click **Settings**. In the **Settings** dialog box, in the **Category** list, select **Fitter Settings**. These optimizations are migrated into the HardCopy II companion revision for placement and timing closure. When designing with a HardCopy II device first, physical synthesis optimizations can be enabled for the HardCopy II device, and these post-fit optimizations are migrated to the Stratix II FPGA revision.

LogicLock™ Regions

The use of LogicLock Regions in the Stratix II FPGA is supported for designs migrating to HardCopy II. However, LogicLock Regions are not passed into the HardCopy II Companion Revision. You can use LogicLock in the HardCopy II design but you must create new LogicLock Regions in the HardCopy II companion revision. In addition, LogicLock Regions in HardCopy II devices can not have their properties set to **Auto Size**. However, Floating LogicLock regions are supported. HardCopy II LogicLock Regions must be manually sized and placed in the floorplan. When LogicLock Regions are created in a HardCopy II device, they start with width and height dimensions set to (**1,1**), and the origin coordinates for placement are at **X1_Y1** in the lower left corner of

the floorplan. You must adjust the size and location of the LogicLock Regions you created in the HardCopy II device before compiling the design.



For information about using LogicLock Regions, refer to the *Quartus II Analyzing and Optimizing Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

PowerPlay Power Analyzer

You can perform power estimation and analysis of your HardCopy II and Stratix II devices using the PowerPlay Early Power Estimator. Use the PowerPlay Power Analyzer for more accurate estimation of your device's power consumption. The PowerPlay Early Power Estimator is available in the Quartus II software version 5.1 and later. The PowerPlay Power Analyzer supports HardCopy II devices in version 6.0 and later of the Quartus II software.



For more information about using the PowerPlay Power Analyzer, refer to the *Quartus II PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook* on the Altera website at **www.altera.com**.

Incremental Compilation

The use of the Quartus II Incremental Compilation in the Stratix II FPGA is supported when migrating a design to a HardCopy II device. Incremental compilation is supported in the Stratix II First design flow or HardCopy II First design flow.

To take advantage of Quartus II Incremental Compilation, organize your design into logical and physical partitions for synthesis and fitting (or place-and-route). Incremental compilation preserves the compilation results and performance of unchanged partitions in your design. This feature dramatically reduces your design iteration time by focusing new compilations only on changed design partitions. New compilation results are then merged with the previous compilation results from unchanged design partitions. You can also target optimization techniques, such as physical synthesis, to specific partitions while leaving other partitions untouched.

In addition, be aware of the following guidelines:

- User partitions and synthesis results are migrated to a companion device.
- LogicLock regions are suggested for user partitions, but are not migrated automatically.

- The first compilation after migration to a companion device requires a full compilation (all partitions are compiled), but subsequent compilations can be incremental if changes to the source RTL are not required. For example, PLL phase changes can be implemented incrementally if the blocks are partitioned.
- The entire design must be migrated between Stratix II and HardCopy II companion devices. The Quartus II software does not support migration of partitions between companion devices.
- Bottom-up Quartus II Incremental Compilation is not supported for HardCopy II devices.
- Physical Synthesis can be run on individual partitions within the originating device only. The resulting optimizations are preserved in the migration to the companion device.



For information about using Quartus II Incremental Compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Maximum Fanout Assignments

This feature is supported beginning in Quartus II 6.1. In order to meet timing, it may be necessary to limit the number of fanouts of a net in your design. You can limit the maximum fanout of a given net by using this feature.

For example, you can use the following Tcl command to enable the maximum fanout setting:

```
set_instance_assignment -name MAX_FANOUT <number>
- to\ <net name>
```

For example, if you want to limit the maximum fanout of net called "m3122_combout_1" to 25, the Tcl command is as follows:

```
set_instance_assignment -name MAX_FANOUT 25 -to\
m3122_combout_1
```

Performing ECOs with Quartus II Engineering Change Management with the Chip Planner

As designs grow larger and larger in density, the need to analyze the design for performance, routing congestion, logic placement, and executing Engineering Change Orders (ECOs) becomes critical. In addition to design analysis, you can use various bottom-up and top-down flows to implement and manage the design. This becomes difficult to manage since ECOs are often implemented as last minute changes to your design.

With the Altera Chip Planner tool, you can shorten the design cycle time significantly. When changes are made to your design as ECOs, you do not have to perform a full compilation in the Quartus II software. Instead, you would make changes directly to the post place-and-route netlist, generate a new programming file, test the revised design by performing a gate-level simulation and timing analysis, and proceed to verify the fix on the system (if you are using a Stratix II FPGA as a prototype). Once the fix has been verified on the Stratix II FPGA, switch to the HardCopy II revision, apply the same ECOs, run the timing analyzer and assembler, perform a revision compare and then run the HardCopy II Netlist Writer for design submission.

There are three scenarios from a migration point of view:

- There are changes which can map one-to-one (that is, the same change can be implemented on each architecture—Stratix II FPGA and HardCopy II).
- There are changes that must be implemented differently on the two architectures to achieve the same result.
- There are some changes that cannot be implemented on both architectures.

The following sections outline the methods for migrating each of these types of changes.

Migrating One-to-One Changes

One-to-one changes are implemented using identical commands in both architectures. In general, such changes include those that affect only I/O cells or PLL cells. Some examples of one-to-one changes are changes such as creating, deleting or moving pins, changing pin or PLL properties, or changing pin connectivity (provided the source and destination of the connectivity changes are I/Os or PLLs). These can be implemented identically on both architectures.

If such changes are exported to Tcl, a direct reapplication of the generated Tcl script (with a minor text edit) on the companion revision should implement the appropriate changes as follows:

- Export the changes from the Change Manager to Tcl.
- Open the generated Tcl script, change the line "project_open <*project*> -revision <*revision*>" to refer to the appropriate companion revision.
- Apply the Tcl script to the companion revision.

A partial list of examples of this type are as follows:

- I/O creation, deletion, and moves
- I/O property changes (for example, I/O standards, delay chain settings, etc.)
- PLL property changes
- Connectivity changes between non-LCELL_COMB atoms (for example, PLL to I/O, DSP to I/O, etc.)

Migrating Changes that Must be Implemented Differently

Some changes must be implemented differently on the two architectures. Changes affecting the logic of the design may fall into this category. Examples are LUTMASK changes, LC_COMB/HSADDER creation and deletion, and connectivity changes not covered in the previous section.

Another example of this would be to have different PLL settings for the Stratix II and the HardCopy II revisions.



For more information about how to use different PLL settings for the Stratix II and HardCopy II Devices, refer to *AN432: Using Different PLL Settings Between Stratix II and HardCopy II Devices*.

Table 4–3 summarizes suggested implementation for various changes.

Table 4–3. Implementation Suggestions for Various Changes (Part 1 of 2)

Change Type	Suggested Implementation
LUTMASK changes	Because a single Stratix II atom may require multiple HardCopy II atoms to implement, it may be necessary to change multiple HardCopy II atoms to implement the change, including adding or modifying connectivity
Make/Delete LC_COMB	If you are using a Stratix II LC_COMB in extended mode (7-LUT) or using a SHARE chain, you must create multiple atoms to implement the same logic functions in HardCopy II. Additionally, the placement of the LC_COMB cell has no meaning in the companion revision as the underlying resources are different.

Table 4–3. Implementation Suggestions for Various Changes (Part 2 of 2)

Change Type	Suggested Implementation
Make/Delete LC_FF	The basic creation and deletion is the same on both architectures. However, as with LC_COMB creation and deletion, the location of an LC_FF in a HardCopy II revision has no meaning in the Stratix II revision and vice versa.
Editing Logic Connectivity	Because a Stratix II LCELL_COMB atom may have to be broken up into several HardCopy II LCELL_COMB atoms, the source or destination ports for connectivity changes may need to be analyzed to properly implement the change in the companion revision.

Changes that Cannot be Migrated

A small set of changes cannot be implemented in the other architecture because they do not make sense in the other architecture. The best example of this occurs when moving logic in a design; because the logic fabric is different between the two architectures, locations in Stratix II make no sense in HardCopy II and vice versa.

Overall Migration Flow

This section outlines the migration flow and the suggested procedure for implementing changes in both revisions to ensure a successful Revision Compare such that the design can be submitted to the HardCopy Design Center.

Preparing the Revisions

The general procedure for migrating changes between devices is the same, whether going from Stratix II to HardCopy II or vice versa. The major steps are as follows:

1. Compile the design on the initial device.
2. Migrate the design from the initial device to the target device in the companion revision.
3. Compile the companion revision.
4. Perform a Revision Compare operation. The two revisions should pass the Revision Compare.

If testing identifies problems requiring ECO changes, equivalent changes can be applied to both Stratix II and HardCopy II revisions, as described in the next section.

Applying ECO Changes

The general flow for applying equivalent changes in companion revisions is as follows:

1. Make changes in one revision using the Chip Planner tools (Chip Planner, Resource Property Editor, and Change Manager), then verify and export these changes. The procedure for doing this is as follows:
   a. Make changes using the Chip Planner tool.
   b. Perform a netlist check using the Check and Save All Netlist Changes command.
   c. Verify correctness using timing analysis, simulation, and prototyping (Stratix II only). If more changes are required, repeat steps a-b.
   d. Export change records from the Change Manager to Tcl scripts, or **.csv** or **.txt** file formats.

      This exported file is used to assist in making the equivalent changes in the companion revision.
2. Open the companion revision in the Quartus II software.
3. Using the exported file, manually reapply the changes using the Chip Planner tool.

   As stated previously, some changes can be reapplied directly to the companion revision (either manually or by applying the Tcl commands), while others require some modifications.
4. Perform a Revision Compare operation. The revisions should now match once again.
5. Verify the correctness of all changes (you may need to run timing analysis).
6. Run the HardCopy II Assembler and the HardCopy II Netlist Writer for design submission along with handoff files.

The Tcl command for running the HardCopy II Assembler is as follows:

```
execute_module -tool asm -args "--
read_settings_files=\ off --write_settings_files=off"
```

The Tcl command for the HardCopy II Netlist Writer is as follows:

```
execute_module -tool cdb \
       -args "--generate_hardcopyii_files"\
```



For more information about using Chip Planner, refer to the *Quartus II Engineering Change Management with Chip Planner* chapter in volume 3 of the *Quartus II Handbook* at **www.altera.com**.

Formal Verification of Stratix II and HardCopy II Revisions

Third-party formal verification software is available for your HardCopy II design. Cadence Encounter Conformal verification software is used for Stratix II and HardCopy II families, as well as several other Altera product families.

To use the Conformal software with the Quartus II software project for your Stratix II and HardCopy II design revisions, you must enable the **EDA Netlist Writer.** It is necessary to turn on the EDA Netlist Writer so it can generate the necessary netlists and command files needed to run the Conformal software. To automatically run the EDA Netlist Writer during the compile of your Stratix II and HardCopy II design revisions, perform the following steps:

1. On the Assignment menu, click **EDA Tool Settings**. The **Settings** dialog box displays.
2. In the **EDA Tool Settings** list, select **Formal Verification**, and in the **Tool name** list, select **Conformal LEC**.
3. Compile your Stratix II and Hardcopy II design revisions, with both the EDA Tool Settings and the Conformal LEC turned on so the EDA Netlist Writer automatically runs.

The Quartus II EDA Netlist Writer produces one netlist for Stratix II when it is run on that revision, and generates a second netlist when it runs on the HardCopy II revision. You can compare your Stratix II post-compile netlist to your RTL source code using the scripts generated by the EDA Netlist Writer. Similarly, you can compare your HardCopy II post-compile netlist to your RTL source code with scripts provided by the EDA Netlist Writer.



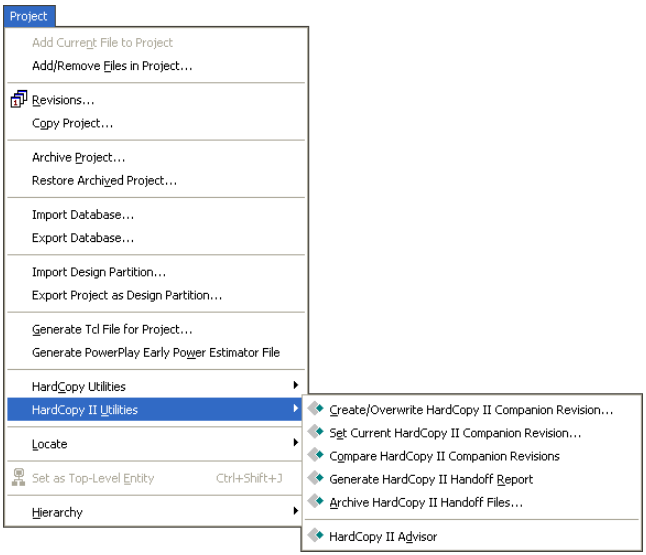
For more information about using the Cadence Encounter Conformal verification software, refer to the *Cadence Encounter Conformal Support* chapter in volume 3 of the *Quartus II Handbook*.

HardCopy II Utilities Menu

The **HardCopy II Utilities** menu in the Quartus II software is shown Figure 4–9. To access this menu, on the Project menu, click **HardCopy II Utilities**. This menu contains the main functions you use to develop your HardCopy II design and Stratix II FPGA prototype companion revision. From the HardCopy II Utilities menu, you can:

- Create or update HardCopy II companion revisions
- Set which HardCopy II companion revision is the current revision
- Generate a HardCopy II Handoff Report for design reviews
- Archive HardCopy II Handoff Files for submission to the HardCopy Design Center
- Compare the companion revisions for functional equivalence
- Track your design progress using the HardCopy II Advisor

Figure 4–9. HardCopy II Utilities Menu



Each of the features within **HardCopy II Utilities** is summarized in Table 4–4. The process for using each of these features is explained in the following sections.

Table 4–4. HardCopy II Utilities Menu Options

Menu	Description	Applicable Design Revision	Restrictions
Create/Overwrite HardCopy II Companion Revision	Create a new companion revision or update an existing companion revision for your Stratix II and HardCopy II design.	Stratix II prototype design and HardCopy II Companion Revision	• Must disable Auto Device selection • Must set a Stratix II device and a HardCopy II companion device
Set Current HardCopy II Companion Revision	Specify which companion revision to associate with current design revision.	Stratix II prototype design and HardCopy II Companion Revision	Companion Revision must already exist
Compare HardCopy II Companion Revisions	Compares the Stratix II design revision with the HardCopy II companion design revision and generates a report.	Stratix II prototype design and HardCopy II Companion Revision	Compilation of both revisions must be complete
Generate HardCopy II Handoff Report	Generate a report containing important design information files and messages generated by the Quartus II compile	Stratix II prototype design and HardCopy II Companion Revision	• Compilation of both revisions must be complete • Compare HardCopy II Companion Revisions must have been executed
Archive HardCopy II Handoff Files	Generate a Quartus II Archive File specifically for submitting the design to the HardCopy Design Center. Similar to the HardCopy Files Wizard for HardCopy Stratix and APEX.	HardCopy II Companion Revision	• Compilation of both revisions must be completed • Compare HardCopy II Companion Revisions must have been executed • Generate HardCopy Handoff Report must have been executed
HardCopy II Advisor	Open an Advisor, similar to the Resource Optimization Advisor, helping you through the steps of creating a HardCopy II project.	Stratix II prototype design and HardCopy II Companion Revision	None

Companion Revisions

HardCopy II designs follow a different development flow in the Quartus II software compared with previous HardCopy families. You can create multiple revisions of your Stratix II prototype design, but you can also create separate revisions of your design for a HardCopy II device. The Quartus II software creates specific HardCopy II design

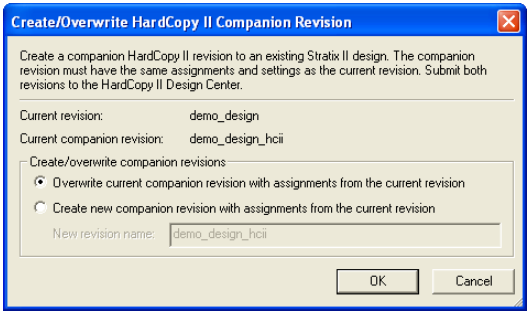
revisions of the project in conjunction to the regular project revisions. These parallel design revisions for HardCopy II devices are called companion revisions.



Although you can create multiple project revisions, Altera recommends that you maintain only one Stratix II FPGA revision once you have created the HardCopy II companion revision.

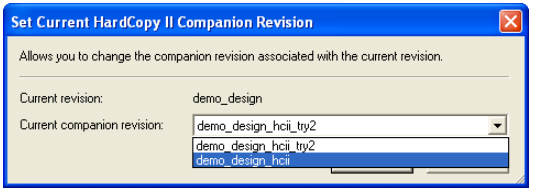
When you have successfully compiled your Stratix II prototype FPGA, you can create a HardCopy II companion revision of your design and proceed with compiling the HardCopy II companion revision. To create a companion revision, on the Project menu, point to HardCopy II Utilities and click **Create/Overwrite HardCopy II Companion Revision**. Use the dialog box to create a new companion revision or overwrite an existing companion revision (Figure 4–10).

Figure 4–10. Create or Overwrite HardCopy II Companion Revision



You can associate only one Stratix II revision to one HardCopy II companion revision. If you created more than one revision or more than one companion revision, set the current companion for the revision you are working on. On the Project menu, point to HardCopy II Utilities and click **Set Current HardCopy II Companion Revision** (Figure 4–11).

Figure 4–11. Set Current HardCopy II Companion Revision

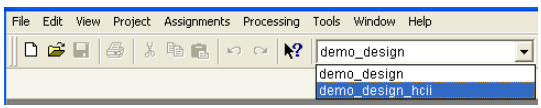


Compiling the HardCopy II Companion Revision

The Quartus II software allows you to compile your HardCopy II design with preliminary timing information. The timing constraints for the HardCopy II companion revision can be the same as the Stratix II design used to create the revision. The Quartus II software contains preliminary timing models for HardCopy II devices and you can gauge how much performance improvement you can achieve in the HardCopy II device compared to the Stratix II FPGA. Altera verifies that the HardCopy II Companion Device timing requirements are met in the HardCopy Design Center.

After you create your HardCopy II companion revision from your compiled Stratix II design, select the companion revision in the Quartus II software design revision drop-down box (Figure 4–12) or from the **Revisions** list. Compile the HardCopy II companion revision. After the Quartus II software compiles your design, you can perform a comparison check of the HardCopy II companion revision to the Stratix II prototype revision.

Figure 4–12. Changing Current Revision



Comparing HardCopy II and Stratix II Companion Revisions

Altera uses the companion revisions in a single Quartus II project to maintain the seamless migration of your design from a Stratix II FPGA to a HardCopy II structured ASIC. This methodology allows you to design with one set of Register Transfer Level (RTL) code to be used in both Stratix II FPGA and HardCopy II structured ASIC, guaranteeing functional equivalency.

When making changes to companion revisions, use the Compare HardCopy II Companion Revisions feature to ensure that your Stratix II design matches your HardCopy II design functionality and compilation settings. To compare companion revisions, on the Project menu, point to HardCopy II Utilities and click **Compare HardCopy II Companion Revisions**.



You must perform this comparison after both Stratix II and HardCopy II designs are compiled in order to hand off the design to Altera's HardCopy Design Center.

The Comparison Revision Summary is found in the Compilation Report and identifies where assignments were changed between revisions or if there is a change in the logic resource count due to different compilation settings.

Generate a HardCopy II Handoff Report

In order to submit a design to the HardCopy Design Center, you must generate a HardCopy II Handoff Report providing important information about the design that you want the HardCopy Design Center to review. To generate the HardCopy II Handoff Report, you must:

- Successfully compile both Stratix II and HardCopy II revisions of your design
- Successfully run the Compare HardCopy II Companion Revisions utility

Once you generate the HardCopy II Handoff Report, you can archive the design using the Archive HardCopy II Handoff Files utility described in "Archive HardCopy II Handoff Files" on page 4–29.

Archive HardCopy II Handoff Files

The last step in the HardCopy II design methodology is to archive the HardCopy II project for submission to the HardCopy Design Center for back-end migration. The HardCopy II archive utility creates a different Quartus II Archive File than the standard Quartus II project archive utility generates. This archive contains only the necessary data from the Quartus II project needed to implement the design in the HardCopy Design Center.

In order to use the **Archive HardCopy II Handoff Files** utility, you must complete the following:

- Compile both the Stratix II and HardCopy II revisions of your design
- Run the Compare HardCopy II Revisions utility
- Generate the HardCopy II Handoff Report

To select this option, on the Project menu, point to HardCopy II Utilities and click **Archive HardCopy II Handoff File** utility.

HardCopy II Advisor

The HardCopy II Advisor provides the list of tasks you should follow to develop your Stratix II prototype and your HardCopy II design. To run the HardCopy II Advisor, on the Project menu, point to HardCopy II Utilities and click **HardCopy II Advisor**. The following list highlights the checkpoints that the HardCopy II Advisor reviews. This list includes the major check points in the design process; it does not show every step in the process for completing your Stratix II and HardCopy II designs:

1. Select a Stratix II device.
2. Select a HardCopy II device.
3. Turn on the **Design Assistant**.
4. Set up timing constraints.
5. Check for incompatible assignments.
6. Compile and check the Stratix II design.
7. Create or overwrite the companion revision.
8. Compile and check the HardCopy II companion results.
9. Compare companion revisions.
10. Generate a Handoff Report.
11. Archive Handoff Files and send to Altera.

The HardCopy II Advisor shows the necessary steps that pertain to your current selected device. The Advisor shows a slightly different view for a design with Stratix II selected as compared to a design with HardCopy II selected.

In the Quartus II software, you can start designing with the HardCopy II device selected first, and build a Stratix II companion revision second. When you use this approach, the HardCopy II Advisor task list adjusts automatically to guide you from HardCopy II development through Stratix II FPGA prototyping, then completes the comparison archiving and handoff to Altera.

When your design uses the Stratix II FPGA as your starting point, Altera recommends following the Advisor guidelines for your Stratix II FPGA until you complete the prototype revision.

When the Stratix II FPGA design is complete, create and switch to your HardCopy II companion revision and follow the Advisor steps shown in that revision until you are finished with the HardCopy II revision and are ready to submit the design to Altera for back-end migration.

Each category in the HardCopy II Advisor list has an explanation of the recommended settings and constraints, as well as quick links to the features in the Quartus II software that are needed for each section. The HardCopy II Advisor displays:

- A green check box when you have successfully completed one of the steps
- A yellow caution sign for steps that must be completed before submitting your design to Altera for HardCopy development
- An information callout for items you must verify



Selecting an item within the HardCopy II flow menu provides a description of the task and recommended action. The view in the HardCopy II Advisor differs depending on the device you select.

Figure 4–13 shows the HardCopy II Advisor with the Stratix II device selected.

Figure 4–13. HardCopy II Advisor with Stratix II Selected

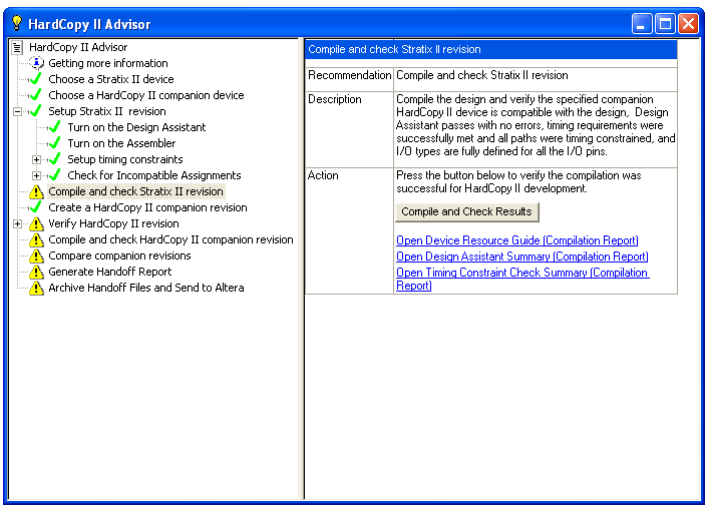
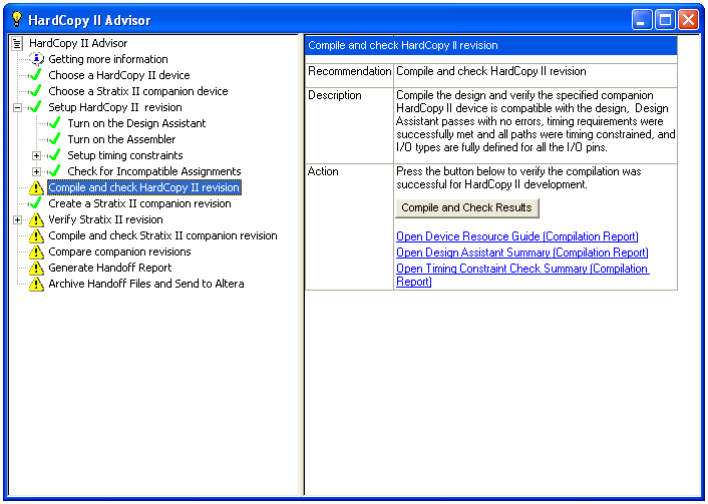


Figure 4–14 shows the HardCopy II Advisor with the HardCopy II device selected.

Figure 4–14. HardCopy II Advisor with HardCopy II Device Selected

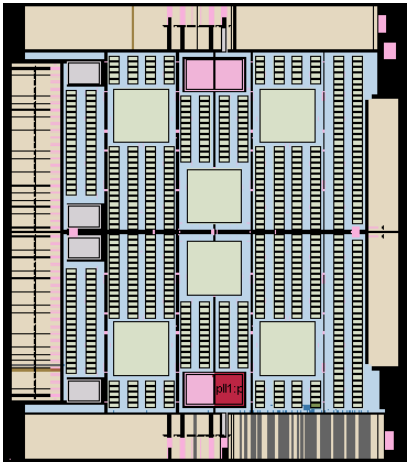


HardCopy II Floorplan View

The Quartus II software displays the preliminary timing closure floorplan and placement of your HardCopy II companion revision. This floorplan shows the preliminary placement and connectivity of all I/O pins, PLLs, memory blocks, HCell macros, and DSP HCell macros. Congestion mapping of routing connections can be viewed using the **Layers Setting** dialog box (in the View menu) settings. This is useful in analyzing densely packed areas of your floorplan that could be reducing the peak performance of your design. The HardCopy Design Center verifies final HCell macro timing and placement to guarantee timing closure is achieved.

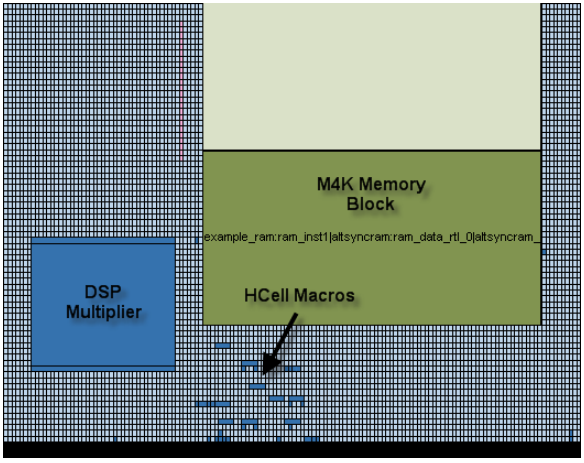
Figure 4–15 shows an example of the HC230F1020 device floorplan.

Figure 4–15. HC230F1020 Device Floorplan



In this small example design, the logic is placed near the bottom edge. You can see the placement of a DSP block constructed of HCell Macros, various logic HCell Macros, and an M4K memory block. A labeled close-up view of this region is shown in Figure 4–16.

Figure 4–16. Close-Up View of Floorplan



The HardCopy Design Center performs final placement and timing closure on your HardCopy II design based on the timing constraints provided in the Stratix II design.



For more information about the HardCopy Design Center's process, refer to the *Back-End Design Flow for HardCopy Series Devices* chapter in volume 1 of the *HardCopy Series Device Handbook*.

HardCopy Stratix Device Support

Altera HardCopy devices provide a comprehensive alternative to ASICs. HardCopy structured ASICs offer a complete solution from prototype to high-volume production, and maintain the powerful features and high-performance architecture of their equivalent FPGAs with the programmability removed. You can use the Quartus II design software to design HardCopy devices in a manner similar to the traditional ASIC design flow and you can prototype with Altera's high density Stratix, APEX 20KC, and APEX 20KE FPGAs before seamlessly migrating to the corresponding HardCopy device for high-volume production.

HardCopy structured ASICs provide the following key benefits:

- Improves performance, on the average, by 40% over the corresponding -6 speed grade FPGA device
- Lowers power consumption, on the average, by 40% over the corresponding FPGA
- Preserves the FPGA architecture and features and minimizes risk
- Guarantees first-silicon success through a proven, seamless migration process from the FPGA to the equivalent HardCopy device
- Offers a quick turnaround of the FPGA design to a structured ASIC device—samples are available in about eight weeks

Altera's Quartus II software has built-in support for HardCopy Stratix devices. The HardCopy design flow in Quartus II software offers the following advantages:

- Unified design flow from prototype to production
- Performance estimation of the HardCopy Stratix device allows you to design systems for maximum throughput
- Easy-to-use and inexpensive design tools from a single vendor
- An integrated design methodology that enables system-on-a-chip designs

This section discusses the following areas:

- How to design HardCopy Stratix and HardCopy APEX structured ASICs using the Quartus II software
- An explanation of what the HARDCOPY_FPGA_PROTOTYPE devices are and how to target designs to these devices
- Performance and power estimation of HardCopy Stratix devices
- How to generate the HardCopy design database for submitting HardCopy Stratix and HardCopy APEX designs to the HardCopy Design Center

Features

Beginning with version 4.2, the Quartus II software contains several powerful features that facilitate design of HardCopy Stratix and HardCopy APEX devices:

- **`HARDCOPY_FPGA_PROTOTYPE` Devices**
  These are virtual Stratix FPGA devices with features identical to HardCopy Stratix devices. You must use these FPGA devices to prototype your designs and verify the functionality in silicon.

- **HardCopy Timing Optimization Wizard**
  Using this feature, you can target your design to HardCopy Stratix devices, providing an estimate of the design's performance in a HardCopy Stratix device.

- **HardCopy Stratix Floorplans and Timing Models**
  The Quartus II software supports post-migration HardCopy Stratix device floorplans and timing models and facilitates design optimization for design performance.

- **Placement Constraints**
  Location and LogicLock constraints are supported at the HardCopy Stratix floorplan level to improve overall performance.

- **Improved Timing Estimation**
  Beginning with version 4.2, the Quartus II software determines routing and associated buffer insertion for HardCopy Stratix designs, and provides the Timing Analyzer with more accurate information about the delays than was possible in previous versions of the Quartus II software. The Quartus II Archive File automatically receives buffer insertion information, which greatly enhances the timing closure process in the back-end migration of your HardCopy Stratix device.

- **Design Assistant**
  This feature checks your design for compliance with all HardCopy device design rules and establishes a seamless migration path in the quickest time.

- **HardCopy Files Wizard**
  This wizard allows you to deliver to Altera the design database and all the deliverables required for migration. This feature is used for HardCopy Stratix and HardCopy APEX devices.

The HardCopy Stratix and HardCopy APEX PowerPlay Early Power Estimator is available on the Altera website at **www.altera.com**.

HARDCOPY_FPGA_PROTOTYPE, HardCopy Stratix and Stratix Devices

You must use the HARDCOPY_FPGA_PROTOTYPE virtual devices available in the Quartus II software to target your designs to the actual resources and package options available in the equivalent post-migration HardCopy Stratix device. The programming file generated for the HARDCOPY_FPGA_PROTOTYPE can be used in the corresponding Stratix FPGA device.

The purpose of the HARDCOPY_FPGA_PROTOTYPE is to guarantee seamless migration to HardCopy by making sure that your design only uses resources in the FPGA that can be used in the HardCopy device after migration. You can use the equivalent Stratix FPGAs to verify the design's functionality in-system, then generate the design database necessary to migrate to a HardCopy device. This process ensures the seamless migration of the design from a prototyping device to a production device in high volume. It also minimizes risk, assures samples in about eight weeks, and guarantees first-silicon success.

HARDCOPY_FPGA_PROTOTYPE devices are only available for HardCopy Stratix devices and are not available for the HardCopy II or HardCopy APEX device families.

Table 4–5 compares HARDCOPY_FPGA_PROTOTYPE devices, Stratix devices, and HardCopy Stratix devices.

Table 4–5. Qualitative Comparison of HARDCOPY_FPGA_PROTOTYPE to Stratix and HardCopy Stratix Devices (Part 1 of 2)

Stratix Device	HARDCOPY_FPGA_PROTOTYPE Device	HardCopy Stratix Device
FPGA	Virtual FPGA	Structured ASIC
FPGA	Architecture identical to Stratix FPGA	Architecture identical to Stratix FPGA

Table 4–5. Qualitative Comparison of HARDCOPY_FPGA_PROTOTYPE to Stratix and HardCopy Stratix Devices (Part 2 of 2)

Stratix Device	HARDCOPY_FPGA_PROTOTYPE Device	HardCopy Stratix Device
FPGA	Resources identical to HardCopy Stratix device	M-RAM resources different than Stratix FPGA in some devices
Ordered through Altera part number	Cannot be ordered, use the Altera Stratix FPGA part number	Ordered by Altera part number

Table 4–6 lists the resources available in each of the HardCopy Stratix devices.

Table 4–6. HardCopy Stratix Device Physical Resources

Device	LEs	ASIC Equivalent Gates (K) *(1)*	M512 Blocks	M4K Blocks	M-RAM Blocks	DSP Blocks	PLLs	Maximum User I/O Pins
HC1S25F672	25,660	250	224	138	2	10	6	473
HC1S30F780	32,470	325	295	171	2 *(2)*	12	6	597
HC1S40F780	41,250	410	384	183	2 *(2)*	14	6	615
HC1S60F1020	57,120	570	574	292	6	18	12	773
HC1S80F1020	79,040	800	767	364	6 *(2)*	22	12	773

Notes to Table 4–6:
(1) Combinational and registered logic do not include DSP blocks, on-chip RAM, or PLLs.
(2) The M-RAM resources for these HardCopy devices differ from the corresponding Stratix FPGA.

For a given device, the number of available M-RAM blocks in HardCopy Stratix devices is identical with the corresponding HARDCOPY_FPGA_PROTOTYPE devices, but may be different from the corresponding Stratix devices. Maintaining the identical resources between HARDCOPY_FPGA_PROTOTYPE and HardCopy Stratix devices facilitates seamless migration from the FPGA to the structured ASIC device.

For more information about HardCopy Stratix devices, refer to the *HardCopy Stratix Device Family Data Sheet* section in volume 1 of the *HardCopy Series Handbook.*

The three devices, Stratix FPGA, HARDCOPY_FPGA_PROTOTYPE, and HardCopy device, are distinct devices in the Quartus II software. The HARDCOPY_FPGA_PROTOTYPE programming files are used in the Stratix FPGA for your design. The three devices are tied together with the same netlist, thus a single SRAM Object File (**.sof)** can be used to achieve the various goals at each stage. The same SRAM Object File is generated

in the HARDCOPY_FPGA_PROTOTYPE design, and is used to program the Stratix FPGA device, the same way that it is used to generate the HardCopy Stratix device, guaranteeing a seamless migration.



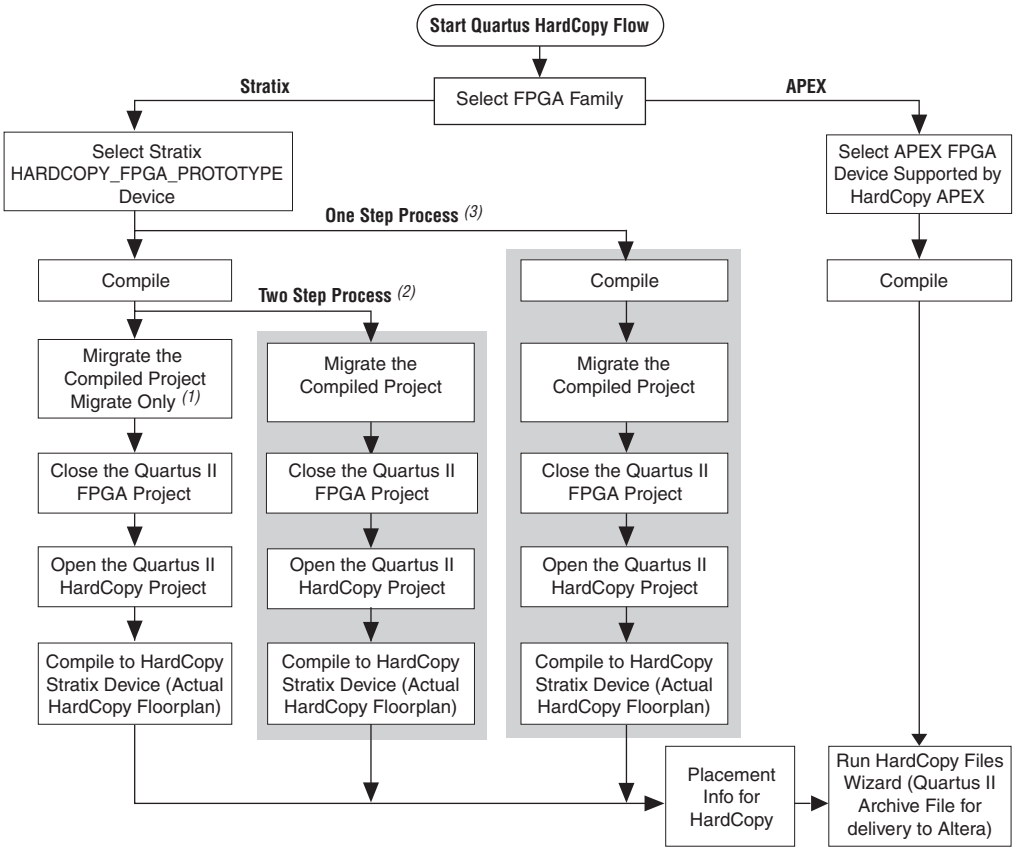
For more information about the SRAM Object File and programming Stratix FPGA devices, refer to the *Programming and Configuration* chapter of the *Introduction to Quartus II Manual*.

HardCopy Design Flow

Figure 4–17 shows a HardCopy design flow diagram. The design steps are explained in detail in the following sections of this chapter. The HardCopy Stratix design flow utilizes the HardCopy Timing Optimization Wizard to automate the migration process into a one-step process. The remainder of this section explains the tasks performed by this automated process.

For a detailed description of the HardCopy Timing Optimization Wizard and HardCopy Files Wizard, refer to "HardCopy Timing Optimization Wizard" on page 4–42 and "Generating the HardCopy Design Database" on page 4–53.

Figure 4–17. HardCopy Stratix and HardCopy APEX Design Flow Diagram



Notes for Figure 4–17:
(1) Migrate-Only Process: The displayed flow is completed manually.
(2) Two-Step Process: Migration and Compilation are done automatically (shaded area).
(3) One-Step Process: Full HardCopy Compilation. The entire process is completed automatically (shaded area).

The Design Flow Steps of the One-Step Process

The following sections describe each step of the full HardCopy compilation (the One Step Process), as shown in Figure 4–17.

Compile the Design for an FPGA

This step compiles the design for a HARDCOPY_FPGA_PROTOTYPE device and gives you the resource utilization and performance of the FPGA.

Migrate the Compiled Project

This step generates the Quartus II Project File (**.qpf**) and the other files required for HardCopy implementation. The Quartus II software also assigns the appropriate HardCopy Stratix device for the design migration.

Close the Quartus FPGA Project

Because you must compile the project for a HardCopy Stratix device, you must close the existing project which you have targeted your design to a HARDCOPY_FPGA_PROTOTYPE device.

Open the Quartus HardCopy Project

Open the Quartus II project that you created in the "Migrate the Compiled Project" step. The selected device is one of the devices from the HardCopy Stratix family that was assigned during that step.

Compile for HardCopy Stratix Device

Compile the design for a HardCopy Stratix device. After successful compilation, the Timing Analysis section of the compilation report shows the performance of the design implemented in the HardCopy device.

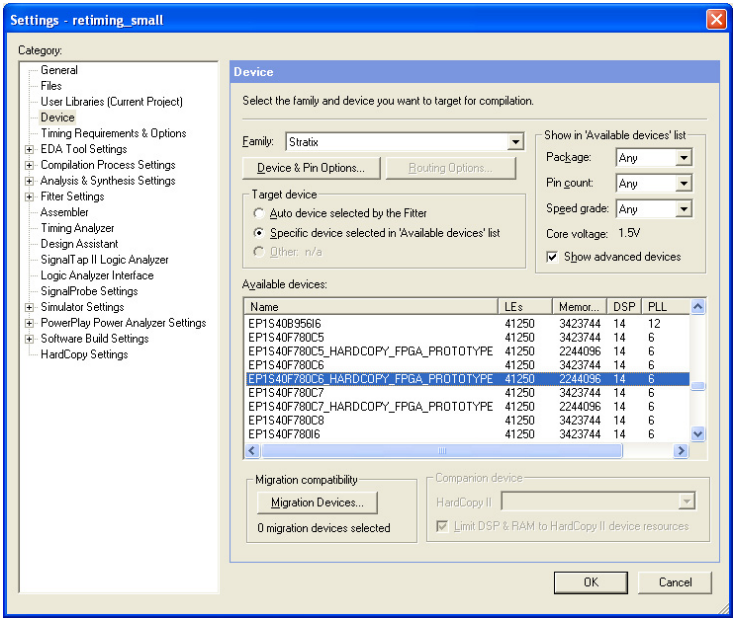
How to Design HardCopy Stratix Devices

This section describes the process for designing for a HardCopy Stratix device using the HARDCOPY_FPGA_PROTOTYPE as your initial selected device. In order to use the HardCopy Timing Optimization Wizard, you must first design with the HARDCOPY_FPGA_PROTOTYPE in order for the design to migrate to a HardCopy Stratix device.

To target a design to a HardCopy Stratix device in the Quartus II software, follow these steps:

1. If you have not yet done so, create a new project or open an existing project.
2. On the Assignments menu, click **Settings**. In the **Category** list, select **Device**.
3. On the **Device** page, in the **Family** list, select **Stratix**. Select the desired HARDCOPY_FPGA_PROTOTYPE device in the **Available Devices** list (Figure 4–18).

Figure 4–18. Selecting a HARDCOPY_FPGA_PROTOTYPE Device



By choosing the HARDCOPY_FPGA_PROTOTYPE device, all the design information, available resources, package option, and pin assignments are constrained to guarantee a seamless migration of your project to the HardCopy Stratix device. The netlist resulting from the HARDCOPY_FPGA_PROTOTYPE device compilation contains information about the electrical connectivity, resources used, I/O placements, and the unused resources in the FPGA device.

4. On the Assignments menu, click **Settings**. In the **Category** list, select **HardCopy Settings** and specify the input transition timing to be modeled for both clock and data input pins. These transition times are used in static timing analysis during back-end timing closure of the HardCopy device.

5. Add constraints to your HARDCOPY_FPGA_PROTOTYPE device, and on the Processing menu, click **Start Compilation** to compile the design.

HardCopy Timing Optimization Wizard

After you have successfully compiled your design in the HARDCOPY_FPGA_PROTOTYPE, you must migrate the design to the HardCopy Stratix device to get a performance estimation of the HardCopy Stratix device. This migration is required before submitting the design to Altera for the HardCopy Stratix device implementation. To perform the required migration, on the Project menu, point to HardCopy Utilities and click **HardCopy Timing Optimization Wizard**.

At this point, you are presented with the following three choices to target the designs to HardCopy Stratix devices (Figure 4–19):

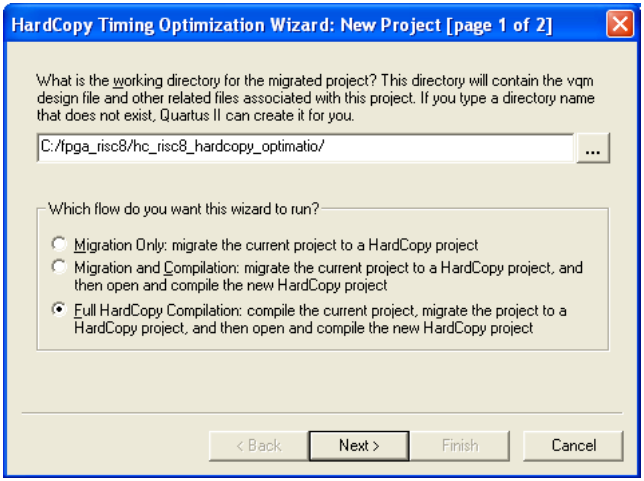
- **Migration Only**: You can select this option after compiling the HARDCOPY_FPGA_PROTOTYPE project to migrate the project to a HardCopy Stratix project.

  You can now perform the following tasks manually to target the design to a HardCopy Stratix device. Refer to"Performance Estimation" on page 4–45 for additional information about how to perform these tasks.
  - Close the existing project
  - Open the migrated HardCopy Stratix project
  - Compile the HardCopy Stratix project for a HardCopy Stratix device

- **Migration and Compilation**: You can select this option after compiling the project. This option results in the following actions:
  - Migrating the project to a HardCopy Stratix project
  - Opening the migrated HardCopy Stratix project and compiling the project for a HardCopy Stratix device

- **Full HardCopy Compilation**: Selecting this option results in the following actions:
  - Compiling the existing HARDCOPY_FPGA_PROTOTYPE project
  - Migrating the project to a HardCopy Stratix project
  - Opening the migrated HardCopy Stratix project and compiling it for a HardCopy Stratix device

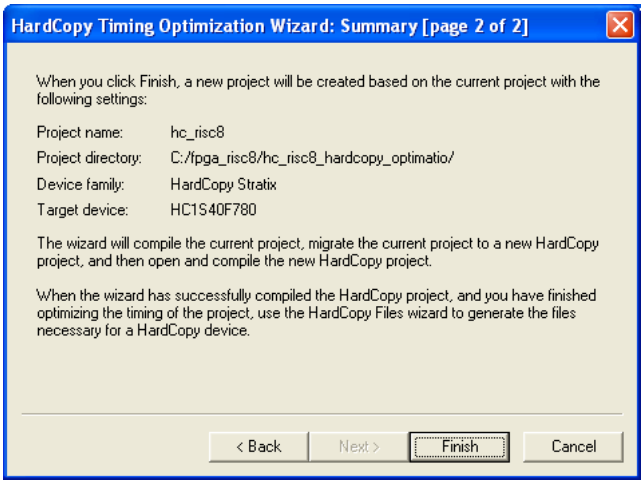
Figure 4–19. HardCopy Timing Optimization Wizard Options



The main benefit of the HardCopy Timing Wizard's three options is flexibility of the conversion process automation. The first time you migrate your HARDCOPY_FPGA_PROTOTYPE project to a HardCopy Stratix device, you may want to use Migration Only, and then work on the HardCopy Stratix project in the Quartus II software. As your prototype FPGA project and HardCopy Stratix project constraints stabilize and you have fewer changes, the Full HardCopy Compilation is ideal for one-click compiling of your HARDCOPY_FPGA_PROTOTYPE and HardCopy Stratix projects.

After selecting the wizard you want to run, the "HardCopy Timing Optimization Wizard: Summary" page shows you details about the settings you made in the Wizard, as shown in Figure 4–20.

Figure 4–20. HardCopy Timing Optimization Wizard Summary Page



When either of the second two options in Figure 4–19 are selected (**Migration and Compilation** or **Full HardCopy Compilation**), designs are targeted to HardCopy Stratix devices and optimized using the HardCopy Stratix placement and timing analysis to estimate performance. For details on the performance optimization and estimation steps, refer to "Performance Estimation" on page 4–45. If the performance requirement is not met, you can modify your RTL source, optimize the FPGA design, and estimate timing until you reach timing closure.

Tcl Support for HardCopy Migration

To complement the GUI features for HardCopy migration, the Quartus II software provides the following command-line executables (which provide the tool command language (Tcl) shell to run the `--flow` Tcl command) to migrate the HARDCOPY_FPGA_PROTOTYPE project to HardCopy Stratix devices:

```
quartus_sh --flow migrate_to_hardcopy <project_name> [-c <revision>] ↵
```

This command migrates the project compiled for the HARDCOPY_FPGA_PROTOTYPE device to a HardCopy Stratix device.

```
quartus_sh --flow hardcopy_full_compile <project_name> [-c <revision>] ↵
```

This command performs the following tasks:

- Compiles the exsisting project for a HARDCOPY_FPGA_PROTOTYPE device.
- Migrates the project to a HardCopy Stratix project.
- Opens the migrated HardCopy Stratix project and compiles it for a HardCopy Stratix device.

Design Optimization and Performance Estimation

The HardCopy Timing Optimization Wizard creates the HardCopy Stratix project in the Quartus II software, where you can perform design optimization and performance estimation of your HardCopy Stratix device.

Design Optimization

Beginning with version 4.2, the Quartus II software supports HardCopy Stratix design optimization by providing floorplans for placement optimization and HardCopy Stratix timing models. These features allows you to refine placement of logic array blocks (LAB) and optimize the HardCopy design further than the FPGA performance. Customized routing and buffer insertion done in the Quartus II software are then used to estimate the design's performance in the migrated device. The HardCopy device floorplan, routing, and timing estimates in the Quartus II software reflect the actual placement of the design in the HardCopy Stratix device, and can be used to see the available resources, and the location of the resources in the actual device.

Performance Estimation

Figure 4–21 illustrates the design flow for estimating performance and optimizing your design. You can target your designs to HARDCOPY_FPGA_PROTOTYPE devices, migrate the design to the HardCopy Stratix device, and get placement optimization and timing estimation of your HardCopy Stratix device.

In the event that the required performance is not met, you can:

- Work to improve LAB placement in the HardCopy Stratix project.

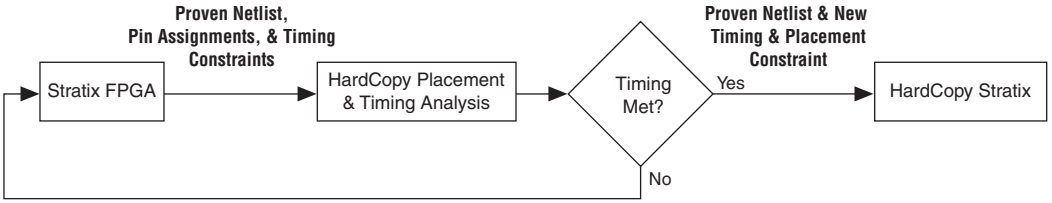
  or

- Go back to the HARDCOPY_FPGA_PROTOTYPE project and optimize that design, modify your RTL source code, repeat the migration to the HardCopy Stratix device, and perform the optimization and timing estimation steps.



On average, HardCopy Stratix devices are 40% faster than the equivalent -6 speed grade Stratix FPGA device. These performance numbers are highly design dependent, and you must obtain final performance numbers from Altera.

Figure 4–21. Obtaining a HardCopy Performance Estimation



To perform Timing Analysis for a HardCopy Stratix device, follow these steps:

1. Open an existing project compiled for a HARDCOPY_FPGA_PROTOYPE device.

2. On the Project menu, point to HardCopy Utilities and click **HardCopy Timing Optimization Wizard**.

3. Select a destination directory for the migrated project and complete the HardCopy Timing Optimization Wizard process.

   On completion of the HardCopy Timing Optimization Wizard, the destination directory created contains the Quartus II project file, and all files required for HardCopy Stratix implementation. At this stage, the design is copied from the HARDCOPY_FPGA_PROTOTYPE project directory to a new directory to perform the timing analysis. This two-project directory structure enables you to move back and forth between the HARDCOPY_FPGA_PROTOTYPE design database and the HardCopy Stratix design database. The Quartus II software creates the <*project name*>**_hardcopy_optimization** directory.

   You do not have to select the HardCopy Stratix device while performing performance estimation. When you run the HardCopy Timing Optimization Wizard, the Quartus II software selects the HardCopy Stratix device corresponding to the specified HARDCOPY_FPGA_PROTOTYPE FPGA. Thus, the information necessary for the HardCopy Stratix device is available from the earlier HARDCOPY_FPGA_PROTOTYPE device selection.

All constraints related to the design are also transferred to the new project directory. You can modify these constraints, if necessary, in your optimized design environment to achieve the necessary timing closure. However, if the design is optimized at the HARDCOPY_FPGA_PROTOTYPE device level by modifying the RTL code or the device constraints, you must migrate the project with the HardCopy Timing Optimization Wizard.



If an existing project directory is selected when the HardCopy Timing Optimization Wizard is run, the existing information is overwritten with the new compile results.

The project directory is the directory that you chose for the migrated project. A snapshot of the files inside the <*project name*>**_hardcopy_optimization** directory is shown in Table 4–7.

Table 4–7. Directory Structure Generated by the HardCopy Timing Optimization Wizard

<*project name*>**_hardcopy_optimization**
- <*project name*>**.qsf**
- <*project name*>**.qpf**
- <*project name*>**.sof**
- <*project name*>**.macr**
- <*project name*>**.gclk**
- **db**
- **hardcopy_fpga_prototype**
  - fpga_<*project name*>**_violations.datasheet**
  - fpga_<*project name*>**_target.datasheet**
  - fpga_<*project name*>**_rba_pt_hcpy_v.tcl**
  - fpga_<*project name*>**_pt_hcpy_v.tcl**
  - fpga_<*project name*>**_hcpy_v.sdo**
  - fpga_<*project name*>**_hcpy.vo**
  - fpga_<*project name*>**_cpld.datasheet**
  - fpga_<*project name*>**_cksum.datasheet**
  - fpga_<*project name*>**.tan.rpt**
  - fpga_<*project name*>**.map.rpt**
  - fpga_<*project name*>**.map.atm**
  - fpga_<*project name*>**.fit.rpt**
  - fpga_<*project name*>**.db_info**
  - fpga_<*project name*>**.cmp.xml**
  - fpga_<*project name*>**.cmp.rcf**
  - fpga_<*project name*>**.cmp.atm**
  - fpga_<*project name*>**.asm.rpt**
  - fpga_<*project name*>**.qarlog**
  - fpga_<*project name*>**.qar**
  - fpga_<*project name*>**.qsf**
  - fpga_<*project name*>**.pin**
  - fpga_<*project name*>**.qpf**
- **db_export**
  - <*project name*>**.map.atm**
  - <*project name*>**.map.hdbx**
  - <*project name*>**.db_info**

4. Open the migrated Quartus II project created in Step 3.

5. Perform a full compilation.

   After successful compilation, the Timing Analysis section of the Compilation Report shows the performance of the design.



Performance estimation is not supported for HardCopy APEX devices in the Quartus II software. Your design can be optimized by modifying the RTL code or the FPGA design and the constraints. You should contact Altera to discuss any desired performance improvements with HardCopy APEX devices.

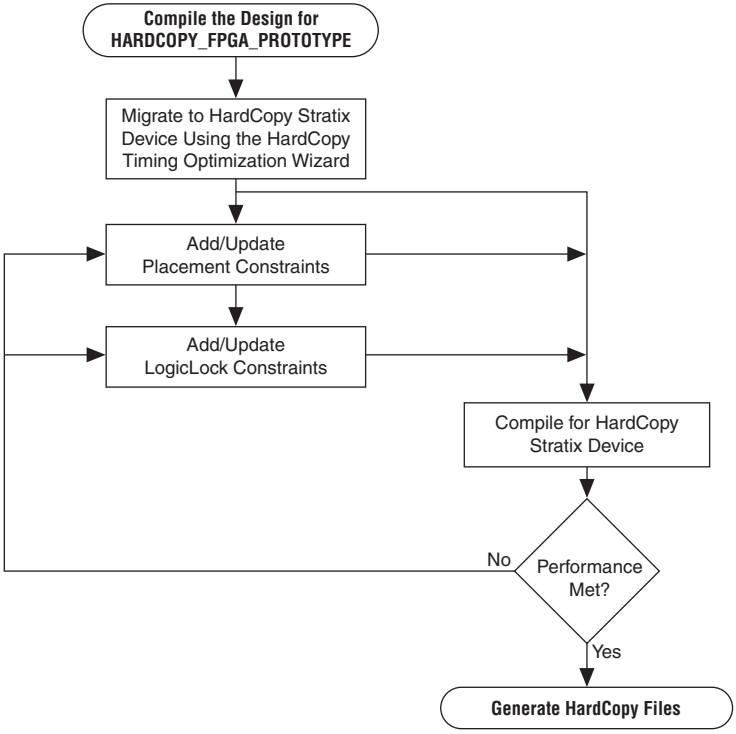
Buffer Insertion

Beginning with version 4.2, the Quartus II software provides improved HardCopy Stratix device timing closure and estimation, to more accurately reflect the results expected after back-end migration. The Quartus II software performs the necessary buffer insertion in your HardCopy Stratix device during the Fitter process, and stores the location of these buffers and necessary routing information in the Quartus II Archive File. This buffer insertion improves the estimation of the Quartus II Timing Analyzer for the HardCopy Stratix device.

Placement Constraints

Beginning with version 4.2, the Quartus II software supports placement constraints and LogicLock regions for HardCopy Stratix devices. Figure 4–22 shows an iterative process to modify the placement constraints until the best placement for the HardCopy Stratix device is achieved.

Figure 4–22. Placement Constraints Flow for HardCopy Stratix Devices



Location Constraints

This section provides information about HardCopy Stratix logic location constraints.

LAB Assignments

Logic placement in HardCopy Stratix is limited to LAB placement and optimization of the interconnecting signals between them. In a Stratix FPGA, individual logic elements (LE) are placed by the Quartus II Fitter into LABs. The HardCopy Stratix migration process requires that LAB contents cannot change after the Timing Optimization Wizard task is done. Therefore, you can only make LAB-level placement optimization and location assignments after migrating the HARDCOPY_FPGA_PROTOTYPE project to the HardCopy Stratix device.

The Quartus II software supports these LAB location constraints for HardCopy Stratix devices. The entire contents of a LAB is moved to an empty LAB when using LAB location assignments. If you want to move the logic contents of LAB A to LAB B, the entire contents of LAB A are moved to an empty LAB B. For example, the logic contents of LAB_X33_Y65 can be moved to an empty LAB at LAB_X43_Y56 but individual logic cell LC_X33_Y65_N1 can not be moved by itself in the HardCopy Stratix Timing Closure Floorplan.

LogicLock Assignments

The LogicLock feature of the Quartus II software provides a block-based design approach. Using this technique you can partition your design and create each block of logic independently, optimize placement and area, and integrate all blocks into the top level design.



To learn more about this methodology, refer to the *Quartus II Analyzing and Optimizing Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

LogicLock constraints are supported when you migrate the project from a HARDCOPY_FPGA_PROTOTYPE project to a HardCopy Stratix project. If the LogicLock region was specified as "Size=Fixed" and "Location=Locked" in the HARDCOPY_FPGA_PROTOTYPE project, it is converted to have "Size=Auto" and "Location=Floating" as shown in the following LogicLock examples. This modification is necessary because the floorplan of a HardCopy Stratix device is different from that of the Stratix device, and the assigned coordinates in the HARDCOPY_FPGA_PROTOTYPE do not match the HardCopy Stratix floorplan. If this modification did not occur, LogicLock assignments would lead to incorrect placement in the Quartus II Fitter. Making the regions auto-size and floating, maintains your LogicLock assignments, allowing you to easily adjust the LogicLock regions as required and lock their locations again after HardCopy Stratix placement.

Example 4–1 and Example 4–2 show two examples of LogicLock assignments.

Example 4–1. LogicLock Region Definition in the HARDCOPY_FPGA_PROTOTYPE Quartus II Settings File

```
set_global_assignment -name LL_HEIGHT 15 -entity risc8 -section_id test
set_global_assignment -name LL_WIDTH 15 -entity risc8 -section_id test
set_global_assignment -name LL_STATE LOCKED -entity risc8 -section_id test
set_global_assignment -name LL_AUTO_SIZE OFF -entity risc8 -section_id test
```

Example 4–2. LogicLock Region Definition in the Migrated HardCopy Stratix Quartus II Settings File

```
set_global_assignment -name LL_HEIGHT 15 -entity risc8 -section_id test
set_global_assignment -name LL_WIDTH 15 -entity risc8 -section_id test
set_global_assignment -name LL_STATE FLOATING -entity risc8 -section_id
test
set_global_assignment -name LL_AUTO_SIZE ON -entity risc8 -section_id test
```

Checking Designs for HardCopy Design Guidelines

When you develop a design with HardCopy migration in mind, you must follow Altera-recommended design practices that ensure a straightforward migration process or the design will not be able to be implemented in a HardCopy device. Prior to starting migration of the design to a HardCopy device, you must review the design and identify and address all the design issues. Any design issues that have not been addressed can jeopardize silicon success.

Altera-Recommended HDL Coding Guidelines

Designing for Altera PLD, FPGA, and HardCopy structured ASIC devices requires certain specific design guidelines and hardware description language (HDL) coding style recommendations be followed.



For more information about design recommendations and HDL coding styles, refer to the *Design Guidelines* section in volume 1 of the *Quartus II Handbook*.

Design Assistant

The Quartus II software includes the Design Assistant feature to check your design against the HardCopy design guidelines. Some of the design rule checks performed by the Design Assistant include the following rules:

- Design should not contain combinational loops
- Design should not contain delay chains
- Design should not contain latches

To use the Design Assistant, you must run Analysis and Synthesis on the design in the Quartus II software. Altera recommends that you run the Design Assistant to check for compliance with the HardCopy design guidelines early in the design process and after every compilation.

Design Assistant Settings

You must select the design rules in the **Design Assistant** page prior to running the design. On the Assignments menu, click **Settings**. In the **Settings** dialog box, in the Category list, select **Design Assistant** and turn on **Run Design Assistant during compilation**. Altera recommends enabling this feature to run the Design Assistant automatically during compilation of your design.

Running Design Assistant

To run Design Assistant independently of other Quartus II features, on the Processing menu, point to Start and click **Start Design Assistant**.

The Design Assistant automatically runs in the background of the Quartus II software when the HardCopy Timing Optimization Wizard is launched, and does not display the Design Assistant results immediately to the display. The design is checked before the Quartus II software migrates the design and creates a new project directory for performing timing analysis.

Also, the Design Assistant runs automatically whenever you generate the HardCopy design database with the HardCopy Files Wizard. The Design Assistant report generated is used by the Altera HardCopy Design Center to review your design.

Reports and Summary

The results of running the Design Assistant on your design are available in the Design Assistant Results section of the Compilation Report. The Design Assistant also generates the summary report in the <*project name*>**hardcopy** subdirectory of the project directory. This report file is titled <*project name*>**_violations.datasheet**. Reports include the settings, run summary, results summary, and details of the results and messages. The Design Assistant report indicates the rule name, severity of the violation, and the circuit path where any violation occurred.



To learn about the design rules and standard design practices to comply with HardCopy design rules, refer to the Quartus II Help and the *HardCopy Series Design Guidelines* chapter in volume 1 of the *HardCopy Series Handbook*.

Generating the HardCopy Design Database

You can use the HardCopy Files Wizard to generate the complete set of deliverables required for migrating the design to a HardCopy device in a single click. The HardCopy Files Wizard asks questions related to the design and archives your design, settings, results, and database files for delivery to Altera. Your responses to the design details are stored in *<project name>*_**hardcopy_optimization***<project name>*.**hps.txt**.

You can generate the archive of the HardCopy design database only after compiling the design to a HardCopy Stratix device. The Quartus II Archive File is generated at the same directory level as the targeted project, either before or after optimization.



The Design Assistant automatically runs when the HardCopy Files Wizard is started.

Table 4–8 shows the archive directory structure and files collected by the HardCopy Files Wizard.

Table 4–8. HardCopy Stratix Design Files Collected by the HardCopy Files Wizard

<*project name*>**_hardcopy_optimization**
- <*project name*>**.flow.rpt**
- <*project name*>**.qpf**
- <*project name*>**.asm.rpt**
- <*project name*>**.blf**
- <*project name*>**.fit.rpt**
- <*project name*>**.gclk**
- <*project name*>**.hps.txt**
- <*project name*>**.macr**
- <*project name*>**.pin**
- <*project name*>**.qsf**
- <*project name*>**.sof**
- <*project name*>**.tan.rpt**

hardcopy
- <*project name*>**.apc**
- <*project name*>**_cksum.datasheet**
- <*project name*>**_cpld.datasheet**
- <*project name*>**_hcpy.vo**
- <*project name*>**_hcpy_v.sdo**
- <*project name*>**_pt_hcpy_v.tcl**
- <*project name*>**_rba_pt_hcpy_v.tcl**
- <*project name*>**_target.datasheet**
- <*project name*>**_violations.datasheet**

hardcopy_fpga_prototype
- fpga_<*project name*>**.asm.rpt**
- fpga_<*project name*>**.cmp.rcf**
- fpga_<*project name*>**.cmp.xml**
- fpga_<*project name*>**.db_info**
- fpga_<*project name*>**.fit.rpt**
- fpga_<*project name*>**.map.atm**
- fpga_<*project name*>**.map.rpt**
- fpga_<*project name*>**.pin**
- fpga_<*project name*>**.qsf**
- fpga_<*project name*>**.tan.rpt**
- fpga_<*project name*>**_cksum.datasheet**
- fpga_<*project name*>**_cpld.datasheet**
- fpga_<*project name*>**_hcpy.vo**
- fpga_<*project name*>**_hcpy_v.sdo**
- fpga_<*project name*>**_pt_hcpy_v.tcl**
- fpga_<*project name*>**_rba_pt_hcpy_v.tcl**
- fpga_<*project name*>**_target.datasheet**
- fpga_<*project name*>**_violations.datasheet**

db_export
- <*project name*>**.db_info**
- <*project name*>**.map.atm**
- <*project name*>**.map.hdbx**

After creating the migration database with the HardCopy Timing Optimization Wizard, you must compile the design before generating the project archive. You will receive an error if you create the archive before compiling the design.

Static Timing Analysis

In addition to performing timing analysis, the Quartus II software also provides all of the requisite netlists and Tcl scripts to perform static timing analysis (STA) using the Synopsys STA tool, PrimeTime. The following files, necessary for timing analysis with the PrimeTime tool, are generated by the HardCopy Files Wizard:

- <*project name*>**_hcpy.vo**—Verilog HDL output format
- <*project name*>**_hpcy_v.sdo**—Standard Delay Format Output File
- <*project name*>**_pt_hcpy_v.tcl**—Tcl script

These files are available in the <*project name*>**\hardcopy** directory. PrimeTime libraries for the HardCopy Stratix and Stratix devices are included with the Quartus II software.

Use the HardCopy Stratix libraries for PrimeTime to perform STA during timing analysis of designs targeted to HARDCOPY_FPGA_PROTOTYPE device.

For more information about static timing analysis, refer to the *Classic Timing Analyzer* and the *Synopsys PrimeTime Support* chapters in volume 3 of the *Quartus II Handbook*.

Early Power Estimation

You can use PowerPlay Early Power Estimation to estimate the amount of power your HardCopy Stratix or HardCopy APEX device will consume. This tool is available on the Altera website. Using the Early Power Estimator requires some knowledge of your design resources and specifications, including:

- Target device and package
- Clock networks used in the design
- Resource usage for LEs, DSP blocks, PLL, and RAM blocks
- High speed differential interfaces (HSDI), general I/O power consumption requirements, and pin counts
- Environmental and thermal conditions

HardCopy Stratix Early Power Estimation

The PowerPlay Early Power Estimator provides an initial estimate of I_{CC} for any HardCopy Stratix device based on typical conditions. This calculation saves significant time and effort in gaining a quick understanding of the power requirements for the device. No stimulus vectors are necessary for power estimation, which is established by the clock frequency and toggle rate in each clock domain.

This calculation should only be used as an estimation of power, not as a specification. The actual I_{CC} should be verified during operation because this estimate is sensitive to the actual logic in the device and the environmental operating conditions.



For more information about simulation-based power estimation, refer to the *Power Estimation and Analysis* Section in volume 3 of the *Quartus II Handbook*.



On average, HardCopy Stratix devices are expected to consume 40% less power than the equivalent FPGA.

HardCopy APEX Early Power Estimation

The PowerPlay Early Power Estimator can be run from the Altera website in the device support section (**http://www.altera.com/support/devices/dvs-index.html**). You cannot open this feature in the Quartus II software.

With the HardCopy APEX PowerPlay Early Power Estimator, you can estimate the power consumed by HardCopy APEX devices and design systems with the appropriate power budget. Refer to the web page for instructions on using the HardCopy APEX PowerPlay Early Power Estimator.



HardCopy APEX devices are generally expected to consume about 40% less power than the equivalent APEX 20KE or APEX 20KC FPGA devices.

Tcl Support for HardCopy Stratix

The Quartus II software also supports the HardCopy Stratix design flow at the command prompt using Tcl scripts.



For details on Quartus II support for Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Targeting Designs to HardCopy APEX Devices

Beginning with version 4.2, the Quartus II software supports targeting designs to HardCopy APEX device families. After compiling your design for one of the APEX 20KC or APEX 20KE FPGA devices supported by a HardCopy APEX device, run the HardCopy Files Wizard to generate the necessary set of files for HardCopy migration.

The HardCopy APEX device requires a different set of design files for migration than HardCopy Stratix. Table 4–9 shows the files collected for HardCopy APEX by the HardCopy Files Wizard.

Table 4–9. HardCopy APEX Files Collected by the HardCopy Files Wizard

<*project name*>**.tan.rpt**
<*project name*>**.asm.rpt**
<*project name*>**.fit.rpt**
<*project name*>**.hps.txt**
<*project name*>**.map.rpt**
<*project name*>**.pin**
<*project name*>**.sof**
<*project name*>**.qsf**
<*project name*>**_cksum.datasheet**
<*project name*>**_cpld.datasheet**
<*project name*>**_hcpy.vo**
<*project name*>**_hcpy_v.sdo**
<*project name*>**_pt_hcpy_v.tcl**
<*project name*>**_rba_pt_hcpy_v.tcl**
<*project name*>**_target.datasheet**
<*project name*>**_violations.datasheet**

Refer to "Generating the HardCopy Design Database" on page 4–53 for information about generating the complete set of deliverables required for migrating the design to a HardCopy APEX device. After you have successfully run the HardCopy Files Wizard, you can submit your design archive to Altera to implement your design in a HardCopy device. You should contact Altera for more information about this process.

Conclusion

The methodology for designing HardCopy Stratix devices using the Quartus II software is the same as that for designing the Stratix FPGA equivalent. You can use the familiar Quartus II software tools and design flow, target designs to HardCopy Stratix devices, optimize designs for higher performance and lower power consumption than the Stratix FPGAs, and deliver the design database for migration to a HardCopy Stratix device. Compatible APEX FPGA designs can migrate to HardCopy APEX after compilation using the HardCopy Files Wizard to archive the design files. Submit the files to the HardCopy Design Center to complete the back-end migration.

Referenced Documents

This chapter references the following documents:

- *AN432: Using Different PLL Settings Between Stratix II and HardCopy II Devices*
- *Back-End Design Flow for HardCopy Series Devices chapter in volume 1 of the HardCopy Series Device Handbook*
- *Cadence Encounter Conformal Support chapter in volume 3 of the Quartus II Handbook*
- *Classic Timing Analyzer chapter in volume 3 of the Quartus II Handbook*
- *Description, Architecture and Features chapter in the HardCopy II Device Family Data Sheet in the HardCopy Series Handbook*
- *Design Guidelines for HardCopy Series Devices chapter of the HardCopy Series Handbook*
- *Design Guidelines Section in volume 1 of the Quartus II Handbook*
- *HardCopy Series Handbook*
- *HardCopy Stratix Device Family Data Sheet section in volume 1 of the HardCopy Series Handbook*
- *Introduction to Quartus II Manual*
- *Introduction to HardCopy II Devices chapter in the HardCopy II Device Family Data Sheet in the HardCopy Series Handbook*
- *Power Estimation and Analysis section in volume 3 of the Quartus II Handbook*
- *Programming and Configuration chapter of the Introduction to Quartus II Manual*
- *Quartus II Analyzing and Optimizing Design Floorplan chapter in volume 2 of the Quartus II Handbook*
- *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook*
- *Quartus II PowerPlay Power Analysis chapter in volume 3 of the Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer chapter in volume 3 of the Quartus II Handbook*
- *Synopsys PrimeTime Support chapter in volume 3 of the Quartus II Handbook*
- *Tcl Scripting chapter in volume 2 of the Quartus II Handbook*

Document Revision History

Table 4–10 shows the revision history for this chapter.

Table 4–10. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 4–58.	Updated for Quartus II version 7.2
May 2007 v7.1.0	• Updated Timing Settings. • Updated TimeQuest. • Added Setting Up the TimeQuest Timing Analzyer. • Added Constraints for Clock Effect Characteristics. • Changed Performing ECOs with Change Manager and Chip Planner title to Performing ECOs with Quartus II Engineering Change Management with the Chip Planner. • Updated Migrating Changes that must be Implemented Differently. • Added Referenced Documents.	Updated for Quartus II version 7.1
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Minor updates for the Quartus II software version 6.1 • Added Performing ECOs with Change Manager and Chip Planner and Overall Migration Flow sections. • Updated Quartus II Software Features Supported for HardCopy II Designs section.	A medium update to the chapter, due to changes in the Quartus II software version 6.1 release; most changes were in the Performing ECOs with Change Manager and Chip Planner and Overall Migration Flow sections.
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
May 2005 v5.0.0	• Chapter 3 was formerly Chapter 2. • Updated for consistency with the *Quartus II Support for HardCopy II Devices* and *Quartus II Support for HardCopy Stratix Devices* chapters in the *HardCopy Series Handbook*.	—
Jan. 2005 v2.1	• Added HardCopy II Device Material.	—
Dec. 2004 v2.1	• Chapter 2 was formerly Chapter 3. • Updates to tables, figures. • New functionality for Quartus II software 4.2	—

Table 4–10. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
June 2004 v2.0	• Updates to tables, figures. • New functionality for Quartus II software 4.1.	—
Feb. 2004 v1.0	Initial release.	—

Section II. Design Guidelines

Today's programmable logic device (PLD) applications have reached the complexity and performance requirements of ASICs. In the development of such complex system designs, good design practices have an enormous impact on your device's timing performance, logic utilization, and system reliability. Designs coded optimally will behave in a predictable and reliable manner, even when re-targeted to different device families or speed grades. This section presents design and coding style recommendations for Altera® devices.

This section includes the following chapters:

- Chapter 5, Design Recommendations for Altera Devices and the Quartus II Design Assistant
- Chapter 6, Recommended HDL Coding Styles



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

5. Design Recommendations for Altera Devices and the Quartus II Design Assistant

QII51006-7.2.0

Introduction

Current FPGA applications have reached the complexity and performance requirements of ASICs. In the development of such complex system designs, good design practices have an enormous impact on your device's timing performance, logic utilization, and system reliability. Well-coded designs behave in a predictable and reliable manner even when re-targeted to different families or speed grades. Good design practices also aid in successful design migration between FPGA and HardCopy® or ASIC implementations for prototyping and production.

For optimal performance, reliability, and faster time-to-market when designing with Altera® devices, you should adhere to the following guidelines:

- Understand the impact of synchronous design practices
- Follow recommended design techniques including hierarchical design partitioning
- Take advantage of the architectural features in the targeted device

This chapter presents design recommendations in these areas, and describes the Quartus® II Design Assistant that can help you check your design for violations of design recommendations.

This chapter contains the following sections:

- "Synchronous FPGA Design Practices" on page 5–2
- "Design Guidelines" on page 5–4
- "Checking Design Violations Using the Design Assistant" on page 5–15
- "Targeting Clock and Register-Control Architectural Features" on page 5–44

For specific HDL coding examples and recommendations, including coding guidelines for targeting dedicated device hardware, such as memory and DSP blocks, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. For information about migrating designs to HardCopy devices, refer to the *Design Guidelines for HardCopy Series Devices* chapter in the *HardCopy Series Handbook*. For guidelines on partitioning a hierarchical design for incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Synchronous FPGA Design Practices

The first step in a good design methodology is to understand the implications of your design practices and techniques. This section outlines some of the benefits of optimal synchronous design practices and the hazards involved in other techniques. Good synchronous design practices can help you meet your design goals consistently. Problems with other design techniques can include reliance on propagation delays in a device, incomplete timing analysis, and possible glitches.

In a synchronous design, a clock signal triggers all events. As long as all of the registers' timing requirements are met, a synchronous design behaves in a predictable and reliable manner for all process, voltage, and temperature (PVT) conditions. You can easily target synchronous designs to different device families or speed grades. In addition, synchronous design practices help ensure successful migration if you plan to migrate your design to a high-volume solution such as Altera HardCopy devices, or if you are prototyping an ASIC.

Fundamentals of Synchronous Design

In a synchronous design, everything is related to the clock signal. On every active edge of the clock (usually the rising edge), the data inputs of registers are sampled and transferred to outputs. Following an active clock edge, the outputs of combinational logic feeding the data inputs of registers change values. This change triggers a period of instability due to propagation delays through the logic as the signals go through a number of transitions and finally settle to new values. Changes happening on data inputs of registers do not affect the values of their outputs until the next active clock edge.

Because the internal circuitry of registers isolates data outputs from inputs, instability in the combinational logic does not affect the operation of the design as long as the following timing requirements are met:

- Before an active clock edge, the data input has been stable for at least the setup time of the register
- After an active clock edge, the data input remains stable for at least the hold time of the register

When you specify all of your clock frequencies and other timing requirements, the Quartus II Classic Timing Analyzer issues actual hardware requirements for the setup times (t_{SU}) and hold times (t_H) for every pin of your design. By meeting these external pin requirements and following synchronous design techniques, you ensure that you satisfy the setup and hold times for all registers within the Altera device.



To meet setup and hold time requirements on all input pins, any inputs to combinational logic that feeds a register should have a synchronous relationship with the clock of the register. If signals are asynchronous, you can register the signals at the input of the Altera device to help prevent a violation of the required setup and hold times.

When the setup or hold time of a register is violated, the output can be set to an intermediate voltage level between the high and low levels, called a metastable state. In this unstable state, small perturbations like noise in power rails can cause the register to assume either the high or low voltage level, resulting in an unpredictable valid state. Various undesirable effects can occur, including increased propagation delays and incorrect output states. In some cases, the output can even oscillate between the two valid states for a relatively long period of time.



For details about timing requirements and analysis in the Quartus II software, refer to the *Quartus II Classic Timing Analyzer* or the *Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*.

Hazards of Asynchronous Design

In the past, designers have often used asynchronous techniques such as ripple counters or pulse generators in programmable logic device (PLD) designs, enabling them to take "short cuts" to save device resources. Asynchronous design techniques have inherent problems such as relying on propagation delays in a device, which can result in incomplete timing constraints and possible glitches and spikes. Because current FPGAs provide many high-performance logic gates, registers, and memory, resource and performance trade-offs have changed. Now it is more important to focus on design practices that help you meet design goals consistently than to save device resources using problematic asynchronous techniques.

Some asynchronous design structures rely on the relative propagation delays of signals to function correctly. In these cases, race conditions can arise where the order of signal changes can affect the output of the logic. PLD designs can have varying timing delays, depending on how the design is placed and routed in the device with each compilation. Therefore, it is almost impossible to determine the timing delay associated with a particular block of logic ahead of time. As devices become faster because of device process improvements, the delays in an asynchronous design may decrease, resulting in a design that does not function as expected. Specific examples are provided in "Design

Guidelines" on page 5–4. Relying on a particular delay also makes asynchronous designs very difficult to migrate to different architectures, devices, or speed grades.

The timing of asynchronous design structures is often difficult or impossible to model with timing assignments and constraints. If you do not have complete or accurate timing constraints, the timing-driven algorithms used by your synthesis and place-and-route tools may not be able to perform the best optimizations, and reported results may not be complete.

Some asynchronous design structures can generate harmful glitches, which are pulses that are very short compared with clock periods. Most glitches are generated by combinational logic. When the inputs of combinational logic change, the outputs exhibit a number of glitches before they settle to their new values. These glitches can propagate through the combinational logic, leading to incorrect values on the outputs in asynchronous designs. In a synchronous design, glitches on the data inputs of registers are normal events that have no negative consequences because the data is not processed until the clock edge.

Design Guidelines

When designing with HDL code, it is important to understand how a synthesis tool interprets different HDL design techniques and what results to expect. Your design techniques can affect logic utilization and timing performance, as well as the design's reliability. This section discusses some basic design techniques that ensure optimal synthesis results for designs targeted to Altera devices while avoiding several common causes of unreliability and instability. Design your combinational logic carefully to avoid potential problems and pay attention to your clocking schemes so you can maintain synchronous functionality and avoid timing problems.

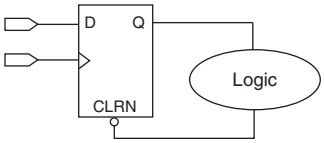
Combinational Logic Structures

Combinational logic structures consist of logic functions that depend only on the current state of the inputs. In Altera FPGAs, these functions are implemented in the look-up tables (LUTs) of the device's architecture, using either logic elements (LEs) or adaptive logic modules (ALMs). For some cases in which combinational logic feeds registers, the register control signals can also be used to implement part of the logic function to save LUT resources. By following the recommendations in this section, you can improve the reliability of your combinational design.

Combinational Loops

Combinational loops are among the most common causes of instability and unreliability in digital designs, and should be avoided whenever possible. In a synchronous design, feedback loops should include registers. Combinational loops generally violate synchronous design principles by establishing a direct feedback loop that contains no registers. For example, a combinational loop occurs when the left-hand side of an arithmetic expression also appears on the right-hand side in HDL code. A combinational loop also occurs when you feed back the output of a register to an asynchronous pin of the same register through combinational logic, as shown in Figure 5–1.

Figure 5–1. Combinational Loop through Asynchronous Control Pin





Use recovery and removal analysis to perform timing analysis on asynchronous ports such as `clear` or `reset` in the Quartus II software. On the Assignments menu, click **Settings**. In the **Settings** dialog box, under **Timing Analysis Settings**, select **Classic Timing Analyzer Settings**. On the **Classic Timing Analyzer Settings** page, click **More Settings**, and turn on the **Enable Recovery/Removal Analysis** option.

Combinational loops are inherently high-risk design structures for the following reasons:

- Combinational loop behavior generally depends on the relative propagation delays through the logic involved in the loop. As discussed, propagation delays can change, which means the behavior of the loop is unpredictable.
- Combinational loops can cause endless computation loops in many design tools. Most tools break open combinational loops to process the design. The various tools used in the design flow may open a given loop in a different manner, processing it in a way that is inconsistent with the original design intent.

Latches

A latch is a small combinational loop that holds the value of a signal until a new value is assigned. Latches can also be inferred from HDL code when you did not intend to use a latch. FPGA architectures are based on

registers. In FPGA devices, latches actually use more logic resources and lead to lower performance than registers. This is different from other device architectures where latches may add less delay and can be implemented with less silicon area than registers.

Latches can cause various difficulties in the design. Although latches are memory elements, they are fundamentally different from registers. When a latch is in feed-through or transparent mode, there is a direct path between the data input and the output. Glitches on the data input can pass through the output. The timing for latches is also inherently ambiguous. For example, when analyzing a design with a D-latch, the software cannot determine whether you intended to transfer data to the output on the leading edge of the clock or on the trailing edge. In many cases, only the original designer knows the full intent of the design; therefore, another designer cannot easily modify the design or reuse the code.

In some cases, your synthesis tool can infer a latch that does not exhibit problems with glitches. Inferring the Altera lpm_latch function ensures that the implementation is glitch-free in Altera architectures. Some third-party synthesis tools list the number of lpm_latch functions that are inferred. When using Quartus II integrated synthesis, these latches are reported in the **User-Specified and Inferred Latches** section of the Compilation Report. If a latch or combinational loop in your design is not listed in this report, it means that it was not inferred as a "safe" latch by the software and is not considered glitch-free.

However, even glitch-free latches may not be analyzed completely during timing analysis. The **Analyze latches as synchronous elements** option in the Quartus II software allows you to treat latches as start and end points for timing analysis (a typical analysis performed in FPGA design tools). With this option turned on, latches are analyzed as registers (with an inverted clock). The Quartus II software does not perform cycle-borrowing analysis, such as that performed by third-party timing analysis tools such as Synopsys PrimeTime.

In addition, latches have a limited support in formal verification tools. Therefore, it is especially important to ensure that you do not use latches when using formal verification.

Altera recommends avoiding using latches to ensure that you can completely analyze and verify the timing performance and reliability of your design.

Delay Chains

Delay chains occur when two or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Inverters are often chained together to add delay. Delay chains are sometimes used to resolve race conditions created by other asynchronous design practices.

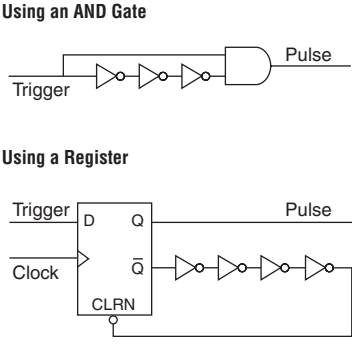
As described earlier in this chapter, delays in PLD designs can change with each place-and-route cycle. Effects such as rise/fall time differences and on-chip variation mean that delay chains, especially those placed on clock paths, can cause significant problems in your design. See "Hazards of Asynchronous Design" on page 5–3 for examples of the kinds of problems that delay chains can cause. Avoid using delay chains to prevent these kind of problems.

In some ASIC designs, delays are used for buffering signals as they are routed around the device. This functionality is not needed in FPGA devices because the routing structure provides buffers throughout the device.

Pulse Generators and Multivibrators

Delay chains are sometimes used to generate either one pulse (pulse generators) or a series of pulses (multivibrators). There are two common methods for pulse generation, as shown in Figure 5–2. These techniques are purely asynchronous and should be avoided.

Figure 5–2. Asynchronous Pulse Generators



In "Using an AND Gate" (Figure 5–2), a trigger signal feeds both inputs of a 2-input AND gate, but the design inverts or adds a delay chain to one of the inputs. The width of the pulse depends on the relative delays of the path that feeds the gate directly and the one that goes through the delay.

This is the same mechanism responsible for the generation of glitches in combinational logic following a change of input values. This technique artificially increases the width of the glitch by using a delay chain.

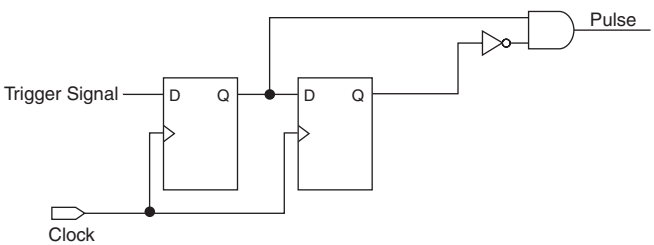
In "Using a Register" (Figure 5–2), a register's output drives the same register's asynchronous reset signal through a delay chain. The register resets itself asynchronously after a certain delay.

The width of pulses generated in this way are difficult for synthesis and place-and-route software to determine, set, or verify. The actual pulse width can only be determined after placement and routing, when routing and propagation delays are known. You cannot reliably determine the width of the pulse when creating HDL code, and it cannot be set by EDA tools. The pulse may not be wide enough for the application under all PVT conditions, and the pulse width changes if you change to a different device. In addition, static timing analysis cannot be used to verify the pulse width, so verification is very difficult.

Multivibrators use a glitch generator to create pulses, together with a combinational loop that turns the circuit into an oscillator. This creates additional problems because of the number of pulses involved. In addition, when the structures generate multiple pulses, they also create a new artificial clock in the design that has to be analyzed by the design tools.

When you must use a pulse generator, use synchronous techniques, as shown in Figure 5–3.

Figure 5–3. Recommended Pulse-Generation Technique



In this design, the pulse width is always equal to the clock period. This pulse generator is predictable, can be verified with timing analysis, and is easily moved to other architectures, devices, or speed grades.

Clocking Schemes

Like combinational logic, clocking schemes have a large effect on your design's performance and reliability. Avoid using internally generated clocks wherever possible because they can cause functional and timing problems in the design. Clocks generated with combinational logic can introduce glitches that create functional problems, and the delay inherent in combinational logic can lead to timing problems. The following sections provide some specific examples and recommendations for avoiding these problems.



Specify all clock relationships in the Quartus II software to allow for the best timing-driven optimizations during fitting and to allow correct timing analysis. Use clock setting assignments on any derived or internal clocks to specify their relationship to the base clock.

Altera recommends using global device-wide, low-skew dedicated routing for all internally-generated clocks, instead of routing clocks on regular routing lines. See "Clock Network Resources" on page 5–44 for a detailed explanation.

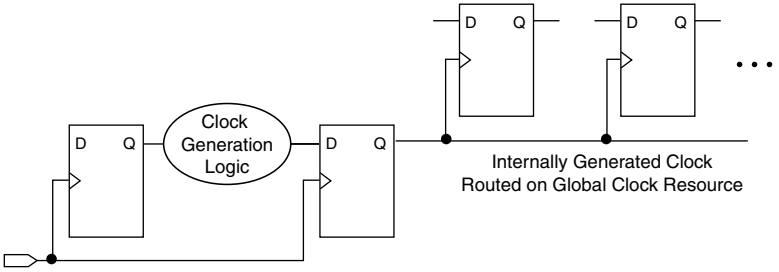
Avoid data transfers between different clocks wherever possible. If a data transfer between different clocks is needed, use FIFO circuitry. You can use the clock uncertainty features in the Quartus II software to compensate for the variable delays between clock domains. Consider setting a Clock Setup Uncertainty and Clock Hold Uncertainty value of 10% to 15% of the clock delay.

Internally Generated Clocks

If you use the output from combinational logic as a clock signal or as an asynchronous reset signal, you should expect to see glitches in your design. In a synchronous design, glitches on data inputs of registers are normal events that have no consequences. However, a glitch or a spike on the clock input (or an asynchronous input) to a register can have significant consequences. Narrow glitches can violate the register's minimum pulse width requirements. Setup and hold times may also be violated if the data input of the register is changing when a glitch reaches the clock input. Even if the design does not violate timing requirements, the register output can change value unexpectedly and cause functional hazards elsewhere in the design.

Because of these problems, always register the output of combinational logic before you use it as a clock signal. See Figure 5–4.

Figure 5–4. Recommended Clock-Generation Technique



Registering the output of combinational logic ensures that the glitches generated by the combinational logic are blocked at the data input of the register.

Divided Clocks

Designs often require clocks created by dividing a master clock. Most Altera FPGAs provide dedicated phase-locked loop (PLL) circuitry for clock division. Using dedicated PLL circuitry can help you to avoid many of the problems that can be introduced by asynchronous clock division logic.

When you must use logic to divide a master clock, always use synchronous counters or state machines. In addition, create your design so that registers always directly generate divided clock signals, as described in "Internally Generated Clocks" on page 5–9, and route the clock on global clock resources. To avoid glitches, you should not decode the outputs of a counter or a state machine to generate clock signals.

Ripple Counters

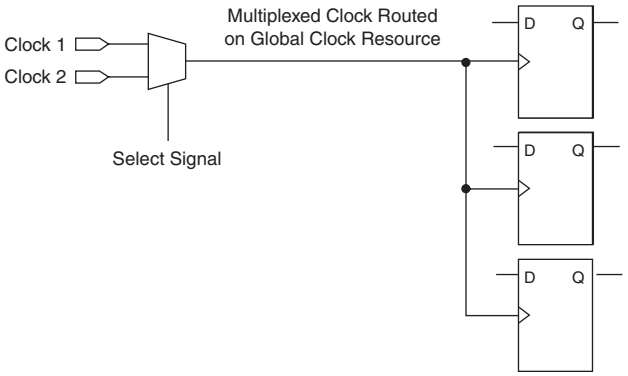
To simplify verification, Altera recommends avoiding ripple counters in your design. In the past, FPGA designers implemented ripple counters to divide clocks by a power of two because the counters are easy to design and may use fewer gates than their synchronous counterparts. Ripple counters use cascaded registers, in which the output pin of each register feeds the clock pin of the register in the next stage. This cascading can cause problems because the counter creates a ripple clock at each stage. These ripple clocks have to be handled properly during timing analysis, which can be difficult and may require you to make complicated timing assignments in your synthesis and place-and-route tools.

Ripple clock structures are often used to make ripple counters out of the smallest amount of logic possible. However, in all Altera devices supported by the Quartus II software, using a ripple clock structure to reduce the amount of logic used for a counter is unnecessary because the device allows you to construct a counter using one logic element per counter bit. Altera recommends that you avoid using ripple counters under any circumstances.

Multiplexed Clocks

Clock multiplexing can be used to operate the same logic function with different clock sources. In these designs, multiplexing selects a clock source, as in Figure 5–5. For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

Figure 5–5. Multiplexing Logic and Clock Sources



Adding multiplexing logic to the clock signal can create the problems addressed in the previous sections, but requirements for multiplexed clocks vary widely depending on the application. Clock multiplexing is acceptable when the clock signal uses global clock routing resources, if the following criteria are met:

- The clock multiplexing logic does not change after initial configuration
- The design uses multiplexing logic to select a clock for testing purposes
- Registers are always reset when the clock switches
- A temporarily incorrect response following clock switching has no negative consequences

If the design switches clocks in real time with no reset signal, and your design cannot tolerate a temporarily incorrect response, you must use a synchronous design so that there are no timing violations on the registers, no glitches on clock signals, and no race conditions or other logical problems. By default, the Quartus II software optimizes and analyzes all possible paths through the multiplexer and between both internal clocks that may come from the multiplexer. This may lead to more restrictive analysis than required if the multiplexer is always selecting one particular clock. If you do not need the more complete analysis, you can assign the output of the multiplexer as a base clock in the Quartus II software, so that all register-register paths are analyzed using that clock.

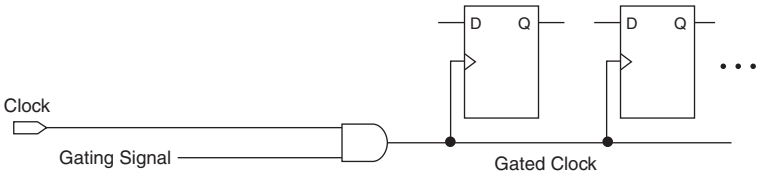
Altera recommends using dedicated hardware to perform clock multiplexing when it is available, instead of using multiplexing logic. For example, you can use the Clock Switchover feature or the Clock Control Block available in certain Altera devices. These dedicated hardware blocks ensure that you use global low-skew routing lines and avoid any possible hold time problems on the device due to logic delay on the clock line.

Refer to the appropriate device data sheet or handbook for device-specific information about clocking structures.

Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls some sort of gating circuitry, as shown in Figure 5–6. When a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

Figure 5–6. Gated Clock



You can use gated clocks to reduce power consumption in some device architectures by effectively shutting down portions of a digital circuit when they are not in use. When a clock is gated, both the clock network and the registers driven by it stop toggling, thereby eliminating their contributions to power consumption. However, gated clocks are not part of a synchronous scheme and therefore can significantly increase the

effort required for design implementation and verification. Gated clocks contribute to clock skew and make device migration difficult. These clocks are also sensitive to glitches, which can cause design failure.

Altera recommends that you use dedicated hardware to perform clock gating rather than using multiplexing logic, if it is available in your target device. For example, you can use the clock control block in newer Altera devices to shut down an entire clock network. Dedicated hardware blocks ensure that you use global routing with low skew and avoid any possible hold time problems on the device due to logic delay on the clock line.

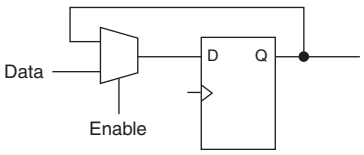
Refer to the appropriate device data sheet or handbook for device-specific information about clocking structures.

From a functional point of view, you can shut down a clock domain in a purely synchronous manner using a synchronous clock enable signal. However, when using a synchronous clock enable scheme, the clock network continues toggling. This practice does not reduce power consumption as much as gating the clock at the source does. In most cases, you should use a synchronous scheme such as those described in "Synchronous Clock Enables". For improved power reduction when gating clocks with logic, refer to "Recommended Clock-Gating Methods" on page 5–14.

Synchronous Clock Enables

To turn off a clock domain in a synchronous manner, use a synchronous clock enable signal. FPGAs efficiently support clock enable signals because there is a dedicated clock enable signal available on all device registers. This scheme does not reduce power consumption as much as gating the clock at the source because the clock network keeps toggling, but it will perform the same function as a gated clock by disabling a set of registers. Insert a multiplexer in front of the data input of every register to either load new data or copy the output of the register (Figure 5–7).

Figure 5–7. Synchronous Clock Enable

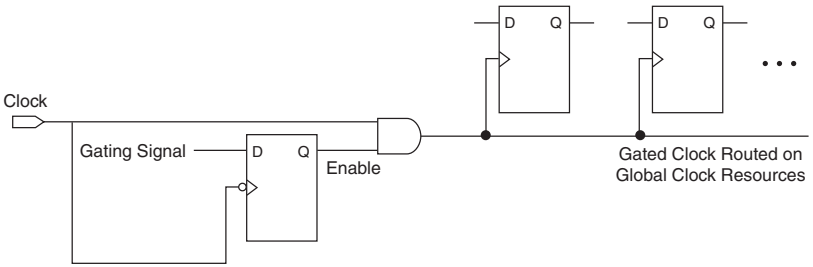


Recommended Clock-Gating Methods

Use gated clocks only when your target application requires power reduction and when gated clocks are able to provide the required reduction in your device architecture. If you must use clocks gated by logic, implement these clocks using the robust clock-gating technique shown in Figure 5–8 and ensure that the gated clock signal uses dedicated global clock routing.

You can gate a clock signal at the source of the clock network, at each register, or somewhere in between. Because the clock network contributes to switching power consumption, gate the clock at the source whenever possible, so you can shut down the entire clock network instead of gating it further along the clock network at the registers.

Figure 5–8. Recommended Clock Gating Technique



In the technique shown in Figure 5–8, a register generates the enable signal to ensure that the signal is free of glitches and spikes. The register that generates the enable signal is triggered on the inactive edge of the clock to be gated (use the falling edge when gating a clock that is active on the rising edge, as shown in Figure 5–8). Using this technique, only one input of the gate that turns the clock on and off changes at a time. This prevents any glitches or spikes on the output. Use an AND gate to gate a clock that is active on the rising edge. For a clock that is active on the falling edge, use an OR gate to gate the clock and register the enable command with a positive edge-triggered register.

When using this technique, pay attention to the duty cycle of the clock and the delay through the logic that generates the enable signal, because the enable signal must be generated in half the clock cycle. This situation might cause problems if the logic that generates the enable command is particularly complex, or if the duty cycle of the clock is severely unbalanced. However, careful management of the duty cycle and logic delay may be an acceptable solution when compared with problems created by other methods of gating clocks.

Ensure that you apply a clock setting to the gated clock in the Quartus II software. As shown in Figure 5–8, apply a clock setting to the output of the AND gate. Otherwise, the timing analyzer may analyze the circuit using the clock path through the register as the longest clock path and the path that skips the register as the shortest clock path, resulting in artificial clock skew.

Checking Design Violations Using the Design Assistant

To improve the reliability, timing performance, and logic utilization of your design, practicing good design methodology and understanding how to avoid design rule violations are important. The Quartus II software provides a tool that automatically checks for design rule violations, and tells you where they occur.

The Design Assistant is a design rule checking tool that allows you to check for design issues early in the design flow. The Design Assistant checks your design for adherence to Altera-recommended design guidelines. You can specify which rules you want the Design Assistant to apply to your design. This is useful if you know that your design violates particular rules that are not critical, so you want to allow these rule violations. The Design Assistant generates design violation reports with clear details about each violation, based on the settings you specified.

The first parts in this section provide an introduction to the Quartus II design flow with Design Assistant, message severity levels, and an explanation about how to set up the Design Assistant. The last parts of the section describe the design rules and the reports generated by the Design Assistant.

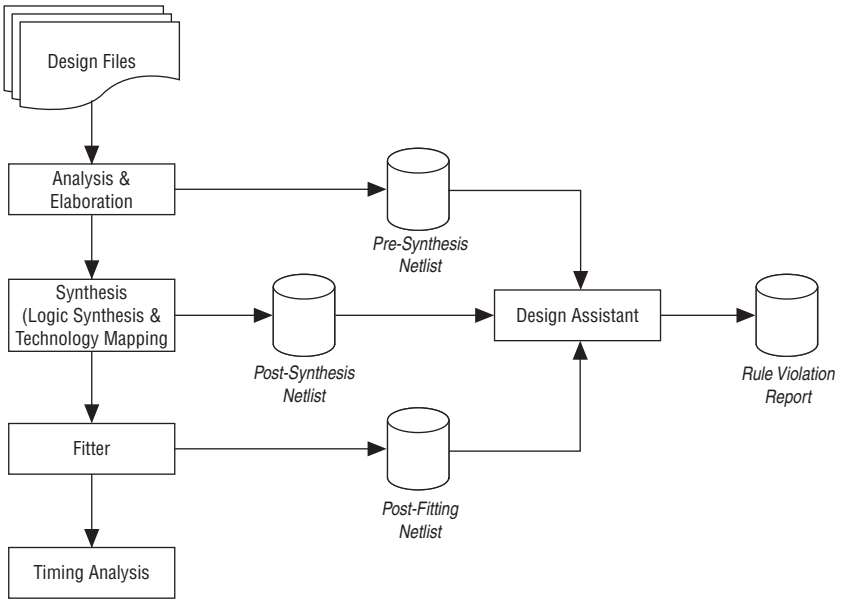
Quartus II Design Flow with the Design Assistant

You can run the Design Assistant after Analysis and Elaboration, Analysis and Synthesis, fitting, or a full compilation. To run the Design Assistant, on the Processing menu, point to Start, and click **Start Design Assistant**.

To set the Design Assistant to run automatically during compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Design Assistant**. Turn on **Run Design Assistant during compilation**. This enables the Design Assistant to perform a post-fitting netlist analysis of your design. The default is to apply all of the rules to your project. But if there are some rules that are unimportant to your design, you can turn off the rules that you do not want the Design Assistant to use. Refer to "The Design Assistant Settings Page" on page 5–17.

Figure 5–9 shows the Quartus II software design flow with the Design Assistant.

Figure 5–9. Quartus II Design Flow with the Design Assistant



The Design Assistant analyzes your design netlist at different stages of the compilation flow and may yield different warnings or errors, even though the netlists are functionally the same. Your pre-synthesis, post-synthesis, and post-fitting netlists may be different due to optimizations performed by the Quartus II software. For example, a warning message in a pre-synthesis netlist may be removed after the netlist has been synthesized into a post-synthesis or post-fitting netlist.

When you run the Design Assistant after running a full compilation or fitting, the Design Assistant performs a post-fitting analysis on the design. When you start the Design Assistant after performing Analysis and Synthesis, the Design Assistant performs post-synthesis analysis on the design. When you start the Design Assistant after performing Analysis and Elaboration, the Design Assistant performs a pre-synthesis analysis on the design. You can also perform pre-synthesis analysis with the Design Assistant using the command-line. You can use `-rtl` option with the **quartus_drc** executable, as shown in the following example:

```
quartus_drc <project_name> --rtl=on ↵
```

The Design Assistant generates warning messages when your design violates design rules, and generates information messages to provide information regarding the rules. The Design Assistant supports all Altera devices supported by the Quartus II software.

The Design Assistant Settings Page

To apply design rules in the Design Assistant, on the Assignments menu, click **Settings**. In the **Settings** dialog box, in the **Category** list, select **Design Assistant**. In the **Design Assistant** page, turn on the rules that you want the Design Assistant to apply during analysis. By default, all of the rules except the finite state machine rules are turned on.

In the **Timing Closure** category, if **Nodes with more than specified number of fan-outs** or **Top nodes with highest fan-out** are turned on, you can use the **High Fan-Out Net Settings** dialog box to specify the number of fan-out a node must have to be reported by the Design Assistant. To open the **High Fan-Out Net Settings** dialog box, in the **Design Assistant** page, in the **Timing Closure** category, select **Nodes with more than specified number of fan-outs** or **Top nodes with highest fan-out**. Click **High Fan-Out Net Settings**.

In the **Clock** category, if you turn on **Clock signal should be a global signal**, you can use the **Global Clock Threshold Settings** dialog box to specify the number of nodes with the highest fan-out which you want the Design Assistant to report. To open the **Global Clock Threshold Settings** dialog box, on the **Design Assistant** page, in the **Clock** category, select **Clock signal should be a global signal**. Click **Global Clock Threshold Settings**.

To specify the maximum number of messages reported by the Design Assistant, on the **Design Assistant** page, click **Report Settings**, and enter the maximum number of violation messages and detail messages to be reported.

Message Severity Levels

The Design Assistant classifies messages and rules using the four severity levels described in Table 5–1. Following Altera guidelines is very important for designs that are migrated to the HardCopy series of devices, therefore the table highlights the impact of a rule violation on a HardCopy migration. Designs that adhere to Altera recommended design guidelines do not produce any messages with critical, high, or medium level of severity.

Table 5–1. Design Assistant Message Severity Levels

Severity Level	Explanation
Critical	A violation of the rule critically affects the reliability of the design. Altera may not be able to implement the design successfully without closely reviewing the violations with the designer for HardCopy device conversions.
High	A violation of the rule affects the reliability of the design. Altera must review the violation before implementing the design for HardCopy device conversions.
Medium	The rule violation may result in implementation complexity which may have an impact for HardCopy device conversions.
Information Only	The rule provides information regarding the design.

Design Assistant Rules

This section describes the Design Assistant rules and details some of the reasons that Altera recommends following certain guidelines. Many of the Design Assistant rules enforce the design guidelines discussed in previous sections of this chapter.

Every rule is represented by a rule ID and has its own severity level. The rule ID is normally used in Tcl commands for rule suppression. The letter in each rule ID corresponds to the group of rules based on the following scheme.

- **A**—Asynchronous design structure rules
- **C**—Clock rules
- **R**—Reset rules
- **S**—Signal race rules
- **T**—Timing closure rules
- **D**—Asynchronous clock domain rules
- **H**—HardCopy rules
- **M**—Finite state machine rules

For example, the rule "Design Should Not Contain Combinational Loops" is the first rule in the asynchronous design structure rules; therefore it is represented by rule ID A101.



The finite state machine rules are applicable only to RTL level verification.

Summary of Rules and IDs

Table 5–2 lists the rules, their rule IDs, and their severity level.

Table 5–2. Summary of Rules and IDs (Part 1 of 2)

Rule ID	Rule Name	Severity Level
A101	Design Should Not Contain Combinational Loops	Critical
A102	Register Output Should Not Drive Its Own Control Signal Directly or through Combinational Logic	Critical
A103	Design Should Not Contain Delay Chains	High
A104	Design Should Not Contain Ripple Clock Structures	Medium
A105	Pulses Should Not Be Implemented Asynchronously	Critical
A106	Multiple Pulses Should Not Be Generated in the Design	Critical
A107	Design Should Not Contain SR Latches	High
A108	Design Should Not Contain Latches	High
A109	Combinational Logic Should Not Directly Drive Write Enable Signal of Asynchronous RAM	Medium
A110	Design Should Not Contain Asynchronous Memory	Medium
C101	Gated Clocks Should Be Implemented According to Altera Standard Scheme	Critical
C102	Logic Cell Should Not Be Used to Generate Inverted Clock	High
C103	Gated Clock Is Not Feeding At Least A Pre-Defined Number Of Clock Ports to Effectively Save Power: <n>	High
C104	Clock Signal Source Should Drive Only Input Clock Ports	Medium
C105	Clock Signal Should Be a Global Signal	High
C106	Clock Signal Source Should Not Drive Registers that Are Triggered by Different Clock Edges	Medium
R101	Combinational Logic Used as a Reset Signal Should Be Synchronized	High
R102	External Reset Should Be Synchronized Using Two Cascaded Registers	Medium
R103	External Reset Should Be Synchronized Correctly	High
R104	Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly	High
R105	Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized	Medium

Table 5–2. Summary of Rules and IDs (Part 2 of 2)

Rule ID	Rule Name	Severity Level
S101	Output Enable and Input of the Same Tri-state Nodes Should Not Be Driven by the Same Signal Source	High
S102	Synchronous Port and Asynchronous Port of the Same Register Should Not Be Driven by the Same Signal Source	High
S103	More Than One Asynchronous Signal Source of the Same Register Should Not Be Driven by the Same Source	High
S104	Clock Port and Any Other Signal Port of the Same Register Should Not Be Driven by the Same Signal Source	High
T101	Nodes with More Than Specified Number of Fan-outs: <n>	Information Only
T102	Top Nodes with Highest Fan-out: <n>	Information Only
D101	Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains	High
D102	Multiple Data Bits Transferred Across Asynchronous Clock Domains Are Synchronized, But Not All Bits May Be Aligned in the Receiving Clock Domain	Medium
D103	Data Bits Are Not Correctly Synchronized When Transferred Between Asynchronous Clock Domains	High
H101	Only One VREF Pin Should Be Assigned to HardCopy Test Pin in an I/O Bank	Medium
H102	A PLL Drives Multiple Clock Network Types	Medium
M101	Data Bits Are Not Synchronized When Transferred to the State Machine of Asynchronous Clock Domains	High
M102	No Reset Signal Defined to Initialize the State Machine	Medium
M103	State Machine Should Not Contain Unreachable State	Medium
M104	State Machine Should Not Contain a Deadlock State	Medium
M105	State Machine Should Not Contain a Dead Transition	Medium

Design Should Not Contain Combinational Loops

Severity Level: Critical
Rule ID: A101

A combinational loop is created by establishing a direct feedback loop on combinational logic that is not synchronized by a register. A combinational loop also occurs when the output of a register is fed back to an asynchronous pin of the same register through combinational logic. Combinational loops are among the most common causes of instability and reliability in your designs, and should be avoided whenever possible. Refer to "Combinational Loops" on page 5–5 for examples of the kinds of problems that combinational loops can cause.

Register Output Should Not Drive Its Own Control Signal Directly or through Combinational Logic

Severity Level: Critical
Rule ID: A102

A combinational loop occurs when you feed back the output of a register to an asynchronous pin of the same register (for example, the register's preset or asynchronous load signal), or the register drives combinational logic that drives one of the control signals on the same register. Combinational loops are among the most common causes of instability and reliability in your designs, and should be avoided whenever possible. Refer to "Combinational Loops" on page 5–5 for examples of the kinds of problems that combinational loops can cause.

Design Should Not Contain Delay Chains

Severity Level: High
Rule ID: A103

Delay chains are created when one or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Delay chains are sometimes used to create intentional delay to resolve race conditions. Delay chains may cause significant problems, because they affect the rise and fall time differences in your design.

This rule applies only for delay chains implemented in logic cells, and is limited to the clock and reset path of your design. This rule does not apply to delay chains in the data path. Altera recommends that you do not instantiate a cell that does not benefit the design, and is used only to delay the signal. Refer to "Delay Chains" on page 5–7 for examples of the kinds of problems that delay chains can cause.

Design Should Not Contain Ripple Clock Structures

Severity Level: Medium
Rule ID: A104

Designs should not contain ripple clock structures. These structures use two or more cascaded registers in which the output of each register feeds the clock pin of the register in the next stage. Cascading structures cause large skew in the output signal because each stage of the structure causes a new clock domain to be defined. The additional clock domains from each stage of the ripple clock are difficult for static timing analysis tools to analyze. Refer to "Ripple Counters" on page 5–10 for examples of the kinds of problems that ripple clock structures can cause.

Pulses Should Not Be Implemented Asynchronously

Severity Level: Critical
Rule ID: A105

There are two common methods for pulse generation:

- Increasing the width of a glitch using a 2-input AND, NAND, OR, or NOR gate, where the source for the two gate inputs are the same, but one of the gate inputs is inverted
- Using a register where the register output drives the register's own asynchronous reset signal through a delay chain (refer to "Delay Chains" on page 5–7 for more details).

These techniques are purely asynchronous and therefore should be avoided. Refer to "Pulse Generators and Multivibrators" on page 5–7 for recommended pulse generation guidelines.

Multiple Pulses Should Not Be Generated in the Design

Severity Level: Critical
Rule ID: A106

A common asynchronous multiple-pulse-generation technique consists of a combinational logic gate in which the inverted output feeds back to one of the inputs of the same gate. This feedback path causes a combinational loop which forces the output to change state, and therefore oscillate. Sometimes multiple pulse generators or multivibrator circuits are built out of a series of cascaded inverters in a structure called a "ring oscillator." Oscillation creates a new artificial clock in your design that is difficult for the Quartus II software to determine, set, or verify.

Structures that generate multiple pulses cause more problems than pulse generators because of the number of pulses involved. In addition, multi-pulse generators also increase the frequency of the design. See "Pulse Generators and Multivibrators" on page 5–7 for recommended pulse generation guidelines.

Design Should Not Contain SR Latches

Severity Level: High
Rule ID: A107

A latch is a combinational loop that holds the value of a signal until a new value is assigned. Combinational loops are hazardous to your design and are the most common causes of instability and reliability. Refer to "Combinational Loops" on page 5–5 for examples of the kinds of problems that combinational loops can cause.

Rule A107 triggers only when your design contains SR latches. An SR latch can cause glitches and ambiguous timing, which complicates the timing analysis of your design. Refer to "Latches" on page 5–5 for details about latches, and for more examples of the kinds of problems that latches can cause.

Design Should Not Contain Latches

Severity Level: High
Rule ID: A108

The Design Assistant generates warnings when it identifies one or more structures as latches.

Refer to "Latches" on page 5–5 for details about latches, and for examples of the kinds of problems that latches can cause.



The difference between A107 ("Design Should Not Contain SR Latches") and A108 is that A107 triggers only when an SR latch is detected. A108 triggers when there's an unidentified latch in your design.

Combinational Logic Should Not Directly Drive Write Enable Signal of Asynchronous RAM

Severity Level: Medium
Rule ID: A109

Altera FPGA devices contain flexible embedded memory structures that can be configured into many different modes. One possible mode is asynchronous RAM. The definition of an asynchronous RAM circuit is one in which the write-enable signal driving into the RAM causes data to be written into it without a clock being required.

You should not use combinational logic to directly drive the write-enable signal of an asynchronous RAM. Any glitches that exist on the write-enable signal can cause the asynchronous RAM to be corrupted. Also, the data and write address ports of the RAM should be stable before the write pulse is asserted, and must remain stable until the write pulse is de-asserted. Because of the limitations to using memory structures in this asynchronous mode, synchronous memories are always preferred. In addition, synchronous memories provide higher design performance.

As a guideline, a register should be used between the combinational logic and the asynchronous RAM, or the asynchronous RAM should be replaced with synchronous memory. Refer to "Hazards of Asynchronous Design" on page 5–3 for examples of the kinds of problems asynchronous techniques can cause.

This rule applies only to device families that support asynchronous RAM.

Design Should Not Contain Asynchronous Memory

Severity Level: Medium
Rule ID: A110

You should avoid using asynchronous memory (for example, asynchronous RAM) in your design, because asynchronous memory can become corrupted by glitches created in the combinational logic that drives the write-enable signal of the memory. Asynchronous memory requires that the data and write address ports of the memory be stable before the write pulse is asserted, and must remain stable until the write pulse is de-asserted. In addition, asynchronous memory has lower performance than synchronous memory.

As a guideline, a register should be used between the combinational logic and the asynchronous RAM, or the asynchronous RAM should be replaced with synchronous memory. Immediately registering both input and output of the RAM improves performance and timing closure. Refer to "Hazards of Asynchronous Design" on page 5–3 for examples of the kinds of problems asynchronous techniques can cause.

This rule applies only to device families that support asynchronous RAM.

Gated Clocks Should Be Implemented According to Altera Standard Scheme

Severity Level: Critical
Rule ID: C101

Clock gating is sometimes used to turn parts of a circuit on and off to reduce the total power consumption of a device. Clock gating is implemented using an enable signal that controls some sort of gating circuitry. The gated clock signal prevents any of the logic driven by it from switching so the logic does not consume any power. For example, when a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive. However, the disadvantage of using this type of circuit is that it can lead to unexpected glitches on the resultant gated clock signal if certain rules are not followed.

Refer to "Gated Clocks" on page 5–12 for examples of the kinds of problems gated clocks can cause. Refer to "Recommended Clock-Gating Methods" on page 5–14 for a recommended clock gating technique. However, when following the recommended clock gating techniques, your design must have a minimum number of fan-outs to meet rule C103, "Gated Clock Is Not Feeding At Least A Pre-Defined Number Of Clock Ports to Effectively Save Power: <n>."

Logic Cell Should Not Be Used to Generate Inverted Clock

Severity Level: High
Rule ID: C102

Your design may require both positive and negative edges of a clock to operate. However, you should not implement an inverter to drive the clock input of a register in your design with a logic cell. Implementing the inverter with a logic cell can lead to clock insertion delay and skew, which is hazardous to your design and can cause problems with the timing closure of the design.

In addition, using a logic cell to implement an inverter is unnecessary. You should use the programmable clock inversion featured in the register to generate the inverted clock signal. Refer to "Clocking Schemes" on page 5–9 for details about different types of clocking methods.

Gated Clock Is Not Feeding At Least A Pre-Defined Number Of Clock Ports to Effectively Save Power: <n>

Severity Level: Medium
Rule ID: C103

Your design can contain an input clock pin that fans out to more than one gated clock. However, to effectively reduce power consumption, Altera recommends that the gated clock feed at least a pre-defined number of clock ports (*n* ports). The default value for *n* is 30. You can set the number of clock ports (*n*) by clicking **Settings** on the Assignments menu. In the **Category** list, select **Design Assistant**. On the **Design Assistant** page, expand the **Clock** category, and turn on **Gated clock is not feeding at least a pre-defined number of clock port to effectively save power: <n>**. Click on the **Gated Clock Settings** button, and in the **Gated Clock Settings** dialog box, set the number of clock ports a gated clock should feed. Refer to "Clocking Schemes" on page 5–9, and "Recommended Clock-Gating Methods" on page 5–14 for proper clock-gating techniques.

Clock Signal Source Should Drive Only Input Clock Ports

Severity Level: Medium
Rule ID: C104

Clock signal sources in a design should drive only input clock ports of registers. Rule C104 triggers when a design contains a clock signal source of a register that connects to port rather than the clock port of another register. Note that if the clock signal source and ports are of the same register, rule S104 "Clock Port and Any Other Signal Port of the Same Register Should Not Be Driven by the Same Signal Source" is triggered instead. Such a design is considered asynchronous and should be avoided. Asynchronous design structures can be hazardous to your design because some of them rely on the relative propagation delays of signals to function correctly, which can result in incomplete timing constraints and possible glitches and spikes. Refer to "Hazards of Asynchronous Design" on page 5–3 for examples of the kinds of problems that asynchronous design structures can cause. Also refer to "Clocking Schemes" on page 5–9 for proper clocking techniques.

This rule does not apply in the following conditions:

- When the clock signal source drives combinational logic that is used as a clock signal, and the combinational logic is implemented according to the Altera standard scheme
- When the clock signal source drives only a clock multiplexer that selects one clock source from a number of different clock sources



This type of multiplexer adds complexity to the timing analysis of a design. You should avoid using the multiplexer in the design.

- Using a clock multiplexer causes the "Gated Clocks Should Be Implemented According to Altera Standard Scheme" rule (C101) to be applied; refer to "Multiplexed Clocks" on page 5–11 for recommended clock multiplexing techniques

Clock Signal Should Be a Global Signal

Severity Level: High
Rule ID: C105

You should ensure that all clock signals in your design use the global clock networks that exist in the target FPGA. Mapping clock signals to use non-dedicated clock networks can negatively affect the performance of your design. A non-global signal can be slower and have larger skew than a global signal because the clock must be distributed using regular FPGA routing resources.

To specify the number of minimum fan-outs that you want the Design Assistant to report, on the **Design Assistant** page, in the **Clock** category, select **Clock signal should be a global signal**. Click **Global Clock Threshold Settings**, and enter the number in the dialog box.

If a design contains more clock signals than are available in the target device, you should consider reducing the number of clock signals in the design, such that only dedicated clock resources are used for clock distribution. However, if the design must use more clock signals than you can specify as global signals, implement the clock signals with the lowest fan-out using regular routing resources. Also, implement the fastest clock signals as global signals. Refer to "Clock Network Resources" on page 5–44 for detailed information about clock resources.

Clock Signal Source Should Not Drive Registers that Are Triggered by Different Clock Edges

Severity Level: Medium
Rule ID: C106

This rule triggers an error message if your design contains a clock signal source that drives the clock inputs of both positive and negative edge-sensitive registers. This error also triggers if your design contains an inverted clock signal that drives the clock inputs of either positive or negative edge-sensitive registers.

These two scenarios can cause an increase in timing requirement complexity and difficulties in design optimization. Also, because those registers are clocked on the different edges, synchronous resetting is impossible. Refer to "Clocking Schemes" on page 5–9 for some specific examples and recommended clocking methods.

Combinational Logic Used as a Reset Signal Should Be Synchronized

Severity Level: High
Rule ID: R101

All combinational logic used to drive reset signals in your design should be synchronized. This means that a register should be placed between the combinational logic that drives reset signal and the input reset pin. Unsynchronized combinational logic can cause glitches and spikes that lead to unintentional reset signals. Synchronizing the combinational logic that drives the reset signal delays the resulting reset signal by an extra clock cycle and avoids unintentional reset. You should consider the extra clock cycle delay when using this method in your design.

Rule R101 does not trigger if the combinational logic used is either a 2-input `AND` or `NOR` that feeds active low reset port, or either a 2-input `OR` or `NAND` that feeds active high reset port.

External Reset Should Be Synchronized Using Two Cascaded Registers

Severity Level: Medium
Rule ID: R102

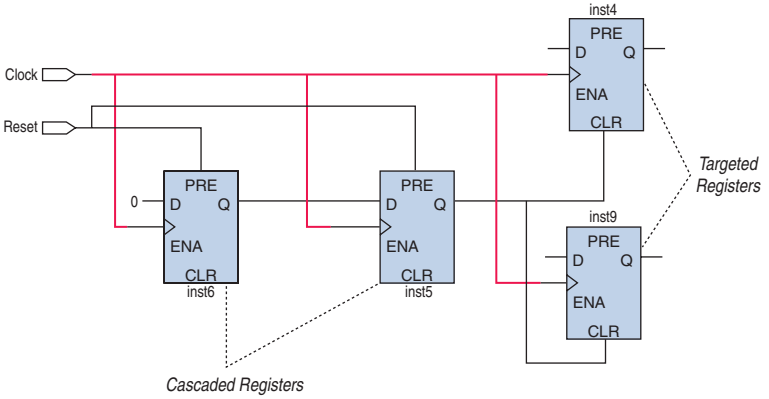
The only way to put your design into a reset state in the absence of a clock signal is to use an asynchronous reset or external reset. However, the asynchronous reset can affect the recovery time of a register, cause design stability problems, and unintentionally reset the state machines in your design to incorrect states.

As a guideline, you can synchronize an external reset signal by using a double-buffer circuit, which consists of two cascaded registers triggered on the same clock edge, and on the same clock domain as the targeted registers.

This rule does not apply in the following two conditions:

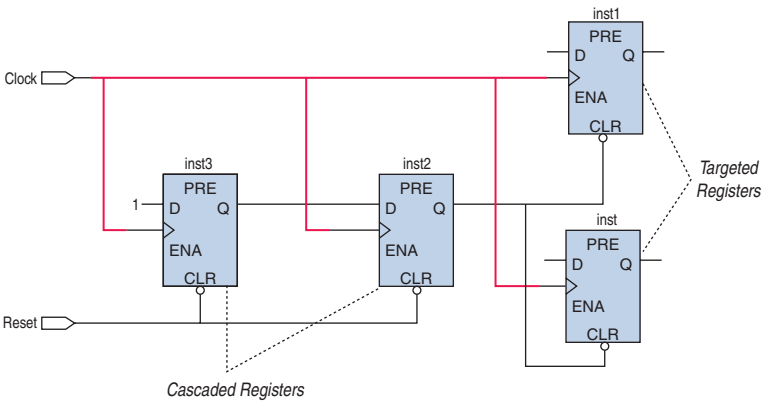
- When the targeted registers use active-high reset ports, and the external reset signal drives the `PRE` ports on the cascaded registers with the input port of the first cascaded registers is fed to GND. Refer to Figure 5–11.

Figure 5–10. Active-Low Reset Ports



- When the targeted registers use active-low reset ports, and the external reset signal drives the CLR ports on the cascaded registers with the input port of the first cascaded registers is fed to V_{CC}. Refer to Figure 5–10.

Figure 5–11. Active-High Reset Ports



External Reset Should Be Synchronized Correctly

Severity Level: High
Rule ID: R103

The only way to put your design into a reset state in the absence of a clock signal is to use an asynchronous reset or external reset. However, the asynchronous reset can affect the recovery time of a register, cause design stability problems, and unintentionally reset the state machines in your design to incorrect states.

As a guideline, you can synchronize an external reset signal by using two cascaded registers. The registers should be triggered on the same clock edge, and on the same clock domain as the targeted registers.

This rule applies when an asynchronous reset or external reset signal is synchronized but fails to follow the recommended guidelines as described in rule R102 ("External Reset Should Be Synchronized Using Two Cascaded Registers"). This violation happens when the external reset is synchronized with only one register, or the cascaded synchronization registers are triggered on different clock edges.



R102 triggers when you don't use two cascaded registers to synchronize the external reset. R103 triggers when the external reset is synchronized but fails to follow the recommended guidelines.

Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly

Severity Level: High
Rule ID: R104

If your design uses an internally generated reset signal generated in one clock domain and used in one or more other asynchronous clock domain, the reset signal should be synchronized. An unsynchronized reset signal can cause metastability issues. To synchronize reset signals across clock domains, use the following guidelines:

- The reset signal should be synchronized with two or more cascading registers in the receiving asynchronous clock domain.
- The cascading registers should be triggered on the same clock edge.

- There should be no logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain. The synchronization registers may sample unintended data due to the glitches caused by the logic.

This rule applies when the internal reset signal is synchronized but fails to follow the recommended guidelines. This happens when the external reset is only synchronized with one register, or the cascaded synchronization registers are triggered on different clock edges, or there is logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain. Synchronizing the reset signal delays the signal by an extra clock cycle. You should consider this delay when using the reset signal in a design.

Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized

Severity Level: Medium
Rule ID: R105

If your design uses an internally generated reset signal that is generated in one clock domain and used in one or more other asynchronous clock domain, the reset signal should be synchronized. An unsynchronized reset signal can cause metastability issues. To synchronize reset signals across clock domains, you should follow guidelines described in Rule R104 ("Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly").

This rule applies when the internally generated reset signal is not being synchronized.

Output Enable and Input of the Same Tri-state Nodes Should Not Be Driven by the Same Signal Source

Severity Level: High
Rule ID: S101

This rule applies when your design contains a tri-state node in which the input and output enable are driven by the same signal source. Signal race occurs between the input and output enable signals of the tri-state when they are propagated simultaneously. Race conditions lead to incorrect design function and unpredictable results. To avoid violation of this rule, the input and output enable of the tri-state should be driven by separate signal sources.

Synchronous Port and Asynchronous Port of the Same Register Should Not Be Driven by the Same Signal Source

Severity Level: High
Rule ID: S102

A purely synchronous design is free of signal race conditions as long as the clock signal is properly distributed and the timing requirements of the registers are met. However, race conditions can occur typically when the synchronous and asynchronous input pins of the register are driven by the same signal source. Race conditions can cause incorrect design function and unpredictable results. Rule S102 triggers when the synchronous and asynchronous pins of a register are driven by the same signal source. Rule S102 does not trigger if the signal source is from a negative-edge sensitive register of the same clock, and if the source register is directly feeding the reset port, provided there is no combinational logic in-between the signal and the register.

More Than One Asynchronous Signal Source of the Same Register Should Not Be Driven by the Same Source

Severity Level: High
Rule ID: S103

To avoid race conditions in your design, Altera recommends that you avoid using the same signal source to drive more than one port on a register. The following ports are affected: `ALOAD`, `ADATA`, `Preset`, and `Clear`.

Clock Port and Any Other Signal Port of the Same Register Should Not Be Driven by the Same Signal Source

Severity Level: High
Rule ID: S104

Any clock signal source in your design should drive only input clock ports of registers. Rule S104 triggers only when your design contains clock signal sources that connect to ports other than the clock ports of the same register. Rule S104 is a sub rule of C104 "Clock Signal Source Should Drive Only Input Clock Ports." Such a design is considered asynchronous and should be avoided. Refer to "Hazards of Asynchronous Design" for examples of the kinds of problems that asynchronous design structures can cause. Refer to "Clocking Schemes" for proper clocking techniques.

Nodes with More Than Specified Number of Fan-outs: <n>

Severity Level: Information Only
Rule ID: T101

This rule reports nodes that have more than a specified number of fan-outs, which can create timing challenges for your design.

To specify the number of fan-outs, on the Assignments menu, click **Settings**. In the **Category** list, select **Design Assistant**. On the **Design Assistant** page, expand the **Timing closure** category by clicking the icon next to **Timing closure**. Turn on **Nodes with more than specified number of fan-outs**. Click **High Fan-Out Net Settings**. In the **High Fan-Out Net Settings** dialog box, enter the number of fan-outs a node must have to be reported by the Design Assistant.

Top Nodes with Highest Fan-out: <n>

Severity Level: Information Only
Rule ID: T102

This rule reports the specified number of nodes with the highest fan-out, which can create timing challenges for your design.

To specify the number of fan-outs, on the Assignments menu, click **Settings**. In the **Category** list, select **Design Assistant**. On the **Design Assistant** page, click the icon next to **Timing closure** to expand the folder. Select **Nodes with more than specified number of fan-outs**. Click **High Fan-out Net Settings**. In the **High Fan-Out Net Settings** dialog box, enter the number of nodes with the highest fan-out to be reported by the Design Assistant.

Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains

Severity Level: High
Rule ID: D101

The data bits transferred between asynchronous clock domains in a design should be synchronized to avoid metastability problems.

If the data bits belong to single-bit data, each data bit should be synchronized with two cascading registers in the receiving asynchronous clock domain, in which the cascaded registers are triggered on the same clock edge. There should be no logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain.

If the data bits belong to multiple-bit data, a handshake protocol should be used to guarantee that all bits of the data bus are stable when the receiving clock domain samples the data. If a handshake protocol is used, only the data bits that act as REQ (request) and ACK (acknowledge) signals should be synchronized. The data bits that belong to multiple-bit data do not need to be synchronized. You can ignore the violation on the data bits that use a handshake protocol.

Multiple Data Bits Transferred Across Asynchronous Clock Domains Are Synchronized, But Not All Bits May Be Aligned in the Receiving Clock Domain

Severity Level: Medium
Rule ID: D102

This rule applies when data bits from a multiple-bit data that are transferred between asynchronous clock domains are synchronized. However, not all data bits may be aligned in the receiving clock domain. Propagation delays may cause skew when the data reaches the receiving clock domain.

If the data bits belong to multiple-bit data and a handshake protocol is used, only the data bits that act as REQ, ACK, or both signals for the transfer should be synchronized with two or more cascading registers in the receiving asynchronous clock domain.

If all of the data bits belong to single-bit data, the synchronization of the data bits does not cause problems in the design.

Data Bits Are Not Correctly Synchronized When Transferred Between Asynchronous Clock Domains

Severity Level: High
Rule ID: D103

The data bits that are transferred between asynchronous clock domains in a design should be synchronized to avoid metastability problems.

If the data bits belong to single-bit data, each data bit should be synchronized with two cascading registers in the receiving asynchronous clock domain. In this case, the cascaded registers are triggered on the same clock edge, and there should be no logic between the output of the transmitting clock domain. The cascaded registers in the receiving asynchronous clock domain.

This rule only applies when the data bits across asynchronous clock domains are synchronized but fail to follow the guidelines.

Only One VREF Pin Should Be Assigned to HardCopy Test Pin in an I/O Bank

Severity Level: Medium
Rule ID: H101

If your design targets a HardCopy APEX™ 20K device, you should not assign more than one VREF pin to a HardCopy test pin in an I/O bank in that targeted device. The assignment of more than one VREF pin to a HardCopy test pin can cause contention of the VREF bus.

You can find the list of the HardCopy test pins that are in each of a HardCopy APEX 20K device's I/O banks in the Messages window, the Design Assistant Messages report, and the Design Assistant HardCopy Test Pins report. You should use this information to ensure that only one VREF pin is assigned to a HardCopy test pin.

However, the Fitter may have assigned the VREF pins to the HardCopy test pins during compilation. To prevent the Fitter from making these assignments during the next compilation, create and assign the VREF pins manually instead of allowing the Fitter to do so automatically.

This rule applies only to designs that target HardCopy APEX 20K devices.

A PLL Drives Multiple Clock Network Types

Severity Level: Medium
Rule ID: H102

A PLL can compensate only one of the clock network types; therefore, the other non-compensated clock network types have a non-zero delay. However, the non-zero delay for the non-compensated clock network types can change between a Stratix device and its corresponding HardCopy Stratix device, or a Stratix II device and its corresponding HardCopy II device.

Therefore, if a Stratix FPGA design relies on the relative offset between the compensated clock network type and the non-compensated clock network types driven by a PLL, an error can occur in the corresponding HardCopy Stratix design because the relative offset in the HardCopy Stratix design may differ from the relative offset in the original Stratix FPGA design.

This rule reports only nodes in a design where a PLL drives multiple clock network types.

Data Bits Are Not Synchronized When Transferred to the State Machine of Asynchronous Clock Domains

Severity Level: High
Rule ID: M101

The data bits that are transferred between asynchronous clock domains in your design should be synchronized to avoid metastability problems. Rule M101 is a state machine specific rule that triggers when input signals of a state machine across asynchronous clock domains are not synchronized or improperly synchronized. Rule M101 applies to state machines only, while the "Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains" rule (D101) and the "Data Bits Are Not Correctly Synchronized When Transferred Between Asynchronous Clock Domains" rule (D103) apply only for data synchronization between registers.

No Reset Signal Defined to Initialize the State Machine

Severity Level: Medium
Rule ID: M102

A finite state machine (FSM) should have a reset signal that initializes to its initial state. A finite state machine without a proper initialization state is susceptible to functional problems and can introduce extra difficulty in analysis, verification, and maintenance.

State Machine Should Not Contain Unreachable State

Severity Level: Medium
Rule ID: M103

An unreachable state is a state that can never be reached from the initial state. Having an unreachable state in your design causes logic redundancy and affects your design functionality. Rule M103 triggers when the initial state cannot traverse to a state through any of the reachable states and transitions.

State Machine Should Not Contain a Deadlock State

Severity Level: Medium
Rule ID: M104

A deadlock state is a state that does not have any transitions to another state except to loop to itself. When the state machine enters a deadlock state, it stays in that state until the state machine is reset. Your design may

have a single state, or a few states forming a deadlock structure. Having a deadlock state in your design leads to design functionality problems, and theoretically may consume more power.

You can change the maximum number of states to be detected as a deadlock structure by clicking **Settings** on the Assignments menu, and in the **Settings** dialog box, in the **Category** list, select **Design Assistant**. In the **Design Assistant** page, click **Finite State Machine Deadlock Settings**. In the **Finite State Machine Deadlock Settings** dialog box, specify the maximum number of states to be reported as a deadlock structure. The default setting is `2`.

State Machine Should Not Contain a Dead Transition

Severity Level: Medium
Rule ID: M105

A dead transition is a redundant transition that never occurs regardless of the event sequence input to the state machine. A dead transition indicates logic redundancy in your design, although it may not affect your design functionality. Rule M105 triggers when the condition required to trigger a transition is not possible.

Enabling and Disabling Design Assistant Rules

You can selectively enable or disable Design Assistant rules on individual nodes by making an assignment in the Assignment Editor, or using the `altera_attribute` synthesis attribute in Verilog HDL or VHDL, or using a Tcl command.



For a list of the types of nodes that allow Design Assistant rule suppression, refer to *Node Types Eligible for Rule Suppression* in the Quartus II Help.



Assignments made with Assignment Editor, the Quartus Settings File (**.qsf**), and Tcl scripts and commands take precedence over assignments made with the `altera_attribute` synthesis attribute. Assignments made to nodes, entities, or instances take precedence over global assignments.

Using the Assignment Editor

You can enable or disable a Design Assistant rule on selected nodes in your design by using the Assignments Editor. You must first compile your design if you have not already done so. On the Assignments menu, click **Assignment Editor**. In the spreadsheet, for the desired node, entity,

or instance, double-click the cell in the Assignment Name column and select **Enable Design Assistant Rule** or **Disable Design Assistant Rule** in the pull-down menu. Then double-click the **Value** cell and type in the Rule ID, or click **Browse** to open the **Design Assistant Rules** dialog box. In the **Design Assistant Rules** dialog box, select the rule you want to enable or disable for that particular node.



You can enable or disable multiple rules by typing more than one Rule ID in the cell, and separating each Rule ID with a comma.

Using Verilog HDL

You can use the `altera_attributes` synthesis attribute in your Verilog HDL code to enable or disable a Design Assistant rule on the selected nodes in your design.

To enable the rule on the selected node, the syntax is as shown in the following example:

```
<entity class> <object> /* synthesis
altera_attribute="enable_da_rule=<ruleID>" */
```

You can enable more than one rule on a selected node as shown in the following example:

```
<entity class> <object> /* synthesis
altera_attribute="enable_da_rule=\"<ruleID>,<ruleID>,
<ruleID>\""*/
```

To disable the rule on the selected node, the syntax is as shown in the following example:

```
<entity class> <object> /* synthesis
altera_attribute="disable_da_rule=<ruleID>" */
```

You can disable more than one rule on a selected node as shown in the following example:

```
<entity class> <object> /* synthesis
altera_attribute="disable_da_rule=\"<ruleID>,
<ruleID>,<ruleID>\""*/
```



When enabling or disabling multiple rules in Verilog HDL, you must separate the Rule ID strings with commas and spaces only, and they must be enclosed with the \" and \" characters.

Using VHDL

You can use the `altera_attributes` synthesis attribute in your VHDL code to enable or disable a Design Assistant rule on the selected nodes in your design.

To enable the rule on the selected node, use the following syntax:

```
attribute altera_attribute : string;attribute
altera_attribute of <object>: <entity class> is
"enable_da_rule=<ruleID>"
```

You can enable more than one rule on a selected node as shown in the following example:

```
attribute altera_attribute : string;attribute
altera_attribute of <object>: <entity class> is
"enable_da_rule=""<ruleID>, <ruleID>, <ruleID>"""
```

To disable the rule on the selected node, use the following syntax:

```
attribute altera_attribute : string;attribute
altera_attribute of <object>: <entity class> is
"disable_da_rule=<ruleID>"
```

You can disable more than one rule on a selected node as shown in the following example:

```
attribute altera_attribute : string;attribute
altera_attribute of <object>: <entity class> is
"disable_da_rule=""<ruleID>, <ruleID>, <ruleID>"""
```



When enabling or disabling multiple rules in VHDL, you must separate the Rule ID strings with commas and spaces only, and they must be enclosed with double quotation mark ("") characters.

Using TCL Commands

To enable a Design Assistant rule on the selected node in your design using Tcl in a script or at a command or Tcl prompt, use the following Tcl command:

```
set_instance_assignment -name enable_da_rule "<rule ID>" -to <design element> ↵
```

To enable more than one rule on a selected node, use the following Tcl command:

```
set_instance_assignment -name enable_da_rule "<rule ID>, <rule ID>" -to <design element> ↵
```

To disable a Design Assistant rule on a selected node in your design using Tcl in a script, or at a command or Tcl prompt, use the following Tcl command:

```
set_instance_assignment -name disable_da_rule "<rule ID>" -to <design element> ↵
```

To disable more than one rule on a selected node, use the following Tcl command:

```
set_instance_assignment -name disable_da_rule "<rule ID>,<rule ID>" -to <design element> ↵
```

Viewing Design Assistant Results

If your design violates a design rule, the Design Assistant generates warning messages and information messages about the violated design rule. The Design Assistant displays these messages in the Messages window, in the Design Assistant Messages report, and in the Design Assistant report files. You can find the Design Assistant report files called <*project_name*>**.drc.rpt** in the <*project_name*> subdirectory of the project directory.

The Design Assistant generates the following reports based on the settings specified in the Design Assistant page:

- Summary Report
- Settings Report
- Detailed Results Report
- Messages Report
- HardCopy Test Pins Report
- Rule Suppression Assignments Report
- Ignored Design Assistant Assignments Report

Summary Report

The Design Assistant Summary report contains summary of the Design Assistant process on a particular project. This includes Design Assistant Status, Revision Name, Top-level Entity, Targeted Family Device, and total number of design violations of the project. The Design Assistant Summary report provides the following information:

- **Design Assistant Status**—the status, end date, and end time of the Design Assistant operation
- **Revision Name**—the revision name specified in the **Revisions** dialog box
- **Top-level Entity Name**—the top-level entity of your design
- **Family**—the device family name specified in the **Device** page of the **Settings** dialog box
- **Total Critical Violations**, **Total High Violations**, **Total Medium Violations**, and **Total Information Only Violations**—the total violations of the rules organized by level, some of which might affect the reliability of the design

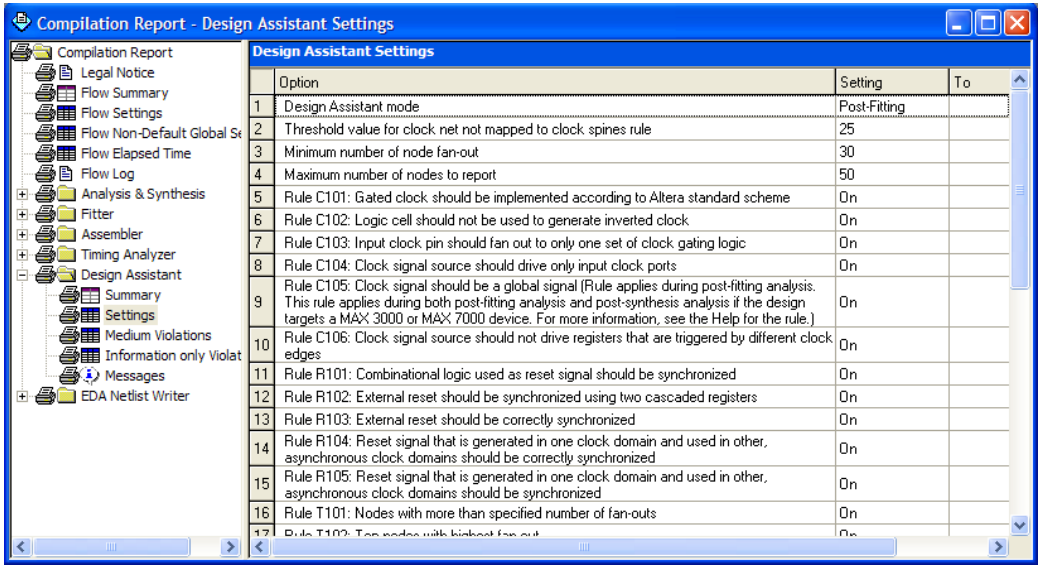


You must first review the violations closely before converting your design for HardCopy devices to achieve a successful conversion.

Settings Report

The Design Assistant Settings report contains a list of enabled Design Assistant rules and options that you specified in the **Design Assistant Settings** page, as shown in Figure 5–12.

Figure 5–12. The Design Assistant Settings Report



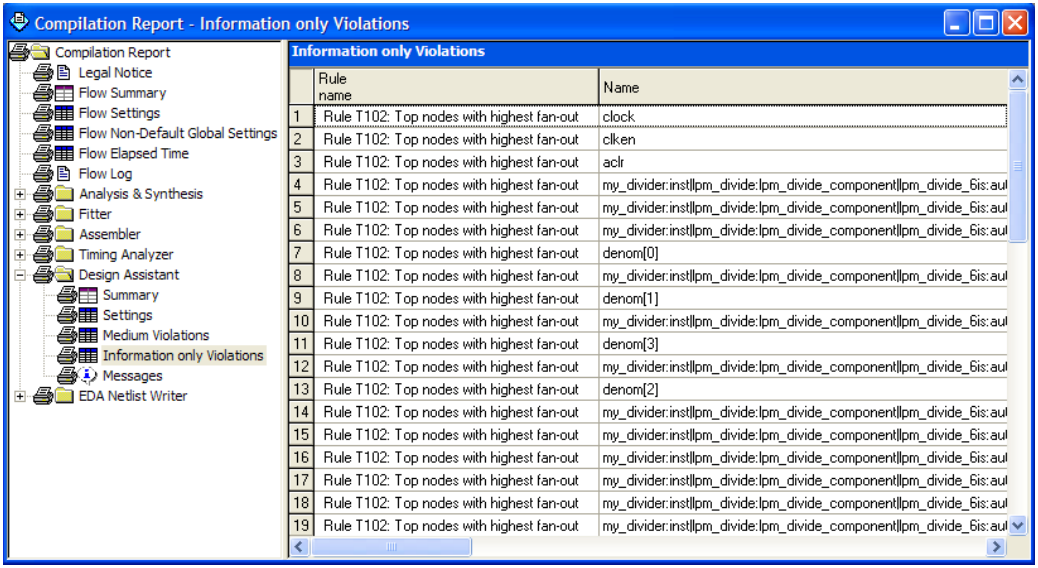
	Option	Setting	To
1	Design Assistant mode	Post-Fitting	
2	Threshold value for clock net not mapped to clock spines rule	25	
3	Minimum number of node fan-out	30	
4	Maximum number of nodes to report	50	
5	Rule C101: Gated clock should be implemented according to Altera standard scheme	On	
6	Rule C102: Logic cell should not be used to generate inverted clock	On	
7	Rule C103: Input clock pin should fan out to only one set of clock gating logic	On	
8	Rule C104: Clock signal source should drive only input clock ports	On	
9	Rule C105: Clock signal should be a global signal (Rule applies during post-fitting analysis. This rule applies during both post-fitting analysis and post-synthesis analysis if the design targets a MAX 3000 or MAX 7000 device. For more information, see the Help for the rule.)	On	
10	Rule C106: Clock signal source should not drive registers that are triggered by different clock edges	On	
11	Rule R101: Combinational logic used as reset signal should be synchronized	On	
12	Rule R102: External reset should be synchronized using two cascaded registers	On	
13	Rule R103: External reset should be correctly synchronized	On	
14	Rule R104: Reset signal that is generated in one clock domain and used in other, asynchronous clock domains should be correctly synchronized	On	
15	Rule R105: Reset signal that is generated in one clock domain and used in other, asynchronous clock domains should be synchronized	On	
16	Rule T101: Nodes with more than specified number of fan-outs	On	

Detailed Results Report

The Detailed Results report contains detailed information of every rule violation including the rule name, the node name, and the fan-out. This report only appears if you specify settings in the **Design Assistant Settings** page. Refer to "The Design Assistant Settings Page" on page 5–17 for more information about how to specify the settings.

Separate Detailed Results reports are generated for critical, high, medium, and information only results. Figure 5–13 shows the **Information Only Violations** report.

Figure 5–13. The Design Detailed Results Report, Information Only



Messages Report

The **Messages** report contains current information, warning, and error messages generated during the Design Assistant process. You can right-click a message in the **Messages** report and click **Help** to display the Quartus II software Help with details about the selected message, or click **Locate** to trace or cross-probe the selected node and locate the source of the violation.

HardCopy Test Pins Report

The HardCopy Test Pins report appears only if you turn on **Run Design Assistant during compilation** in the **Design Assistant** page, and if your design violates the "Only One VREF Pin Should Be Assigned to HardCopy Test Pin in an I/O Bank" rule (H101). The report lists all the HardCopy design rule violations, and also list all of the test pins in the HardCopy device.

Rule Suppression Assignments Report

The Rule Suppression Assignments report contains detailed information about which Design Assistant rules are enabled or disabled, as explained in the "Enabling and Disabling Design Assistant Rules" on page 5–37. The report shows you the following information:

- Assignment—shows the name of the assignment
- Value—identifies the rule
- To—shows the name of the node where the rule is being applied

Ignored Design Assistant Assignments Report

The Ignored Design Assistant Assignments report lists detailed information about the invalid and conflicting rule assignments reported by the Design Assistant. Note that this report is generated only if you specify an invalid rule ID in the rule suppression, or a conflicting rule assignment. The following information appears in the report:

- Assignment—shows the name of the assignment
- Value—identifies the rule
- To—shows the name of the node where the rule is being applied
- Comment—shows why the assignment is being ignored

Targeting Clock and Register-Control Architectural Features

In addition to following general design guidelines, it is important to code your design with the device architecture in mind. FPGAs provide device-wide clocks and register control signals that can improve performance.

Clock Network Resources

Altera FPGAs provide device-wide global clock routing resources and dedicated inputs. You should use the FPGA's low-skew, high fan-out, dedicated routing where available. By assigning a clock input to one of these dedicated clock pins or using a Quartus II logic option to assign global routing, you can take advantage of the dedicated routing available for clock signals.

In ASIC design, balancing the clock delay as it is distributed across the device can be important. Because Altera FPGAs provides device-wide global clock routing resources and dedicated inputs, there is no need to manually balance delays on the clock network.

Altera recommends limiting the number of clocks in your design to the number of dedicated global clock resources available in your FPGA. Clocks feeding multiple locations that do not use global routing may exhibit clock skew across the device that could lead to timing problems.

In addition, when you use combinational logic to generate an internal clock, it adds delays on the clock line. In some cases, delay on a clock line can result in a clock skew greater than the data path length between two registers. If the clock skew is greater than the data delay, the timing parameters of the register (such as hold time requirements) are violated and the design will not function correctly.

Current FPGAs offer increasing numbers of global clocks to address large designs with many clock domains. Many large FPGA devices provide dedicated global clock networks, regional clock networks, and dedicated fast regional clock networks. These clocks are typically organized into a hierarchical clock structure that allows many clocks in each device region with low skew and delay. There are typically a number of dedicated clock pins to drive either the global or regional clock networks and both PLL outputs and internal clocks can drive various clock networks.

To reduce the clock skew within a given clock domain and ensure that hold times are met within that clock domain, assign each clock signal to one of the global high fan-out, low-skew clock networks in the FPGA device. Quartus II automatically uses global routing for high fan-out control signals, PLL outputs, and signals feeding the global clock pins on the device. You can make explicit **Global Signal** logic option settings by turning on the signal logic option settings. On the Assignment menu, click **Assignment Editor**. Use this option when it is necessary to force the software to use the global routing for particular signals.

To take full advantage of these routing resources, the sources of clock signals in a design (input clock pins or internally-generated clocks) should drive only the clock input ports of registers. In older Altera device families (such as FLEX® 10K and ACEX® 1K), if a clock signal feeds the data ports of a register, the signal may not be able to use the dedicated routing, which can lead to decreased performance and clock skew problems. In general, allowing clock signals to drive the data ports of registers is not considered synchronous design, and it can complicate timing analysis. It is not a recommended practice.

Reset Resources

ASIC designs may use local resets to avoid long routing delays on the signal. You should take advantage of the device-wide asynchronous reset pin available on most FPGAs to eliminate these problems. This reset signal provides low-skew routing across the device.

Register Control Signals

Avoid using an asynchronous load signal if the design target device architecture does not include registers with dedicated circuitry for asynchronous loads. Also, avoid using both asynchronous clear and preset if the architecture provides only one of those control signals. APEX devices, for example, directly support an asynchronous clear function, but not a preset or load function. When the target device does not directly support the signals, the place-and-route software must use combinational logic to implement the same functionality. In addition, if you use signals in a priority other than the inherent priority in the device architecture, combinational logic may be required to implement the desired control signals. The combinational logic is less efficient and can cause glitches and other problems; it is best to avoid these implementations.



For Verilog HDL and VHDL examples of registers with various control signals, and information about the inherent priority order of register control signals in Altera device architecture, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Conclusion

Following the design practices outlined in this chapter can help you meet your design goals consistently. Asynchronous design techniques may result in incomplete timing analysis, may clause glitches on data signals, and may rely on propagation delays in a device leading to race conditions and unpredictable results. Taking advantage of the architectural features in your FPGA device can also improve your quality of results.

Referenced Documents

This chapter references the following documents:

- *Design Guidelines for HardCopy Series Devices* chapter in the *HardCopy Series Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*

Document Revision History

Table 5–3 shows the revision history for this chapter.

Table 5–3. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Added restrictions to the rule "External Reset Should Be Synchronized Using Two Cascaded Registers" on page 5–28 • Added Figure 5–11 and 5–10 on page 5–29 • Some changes regarding the Delay Chain rule description (page 5–21) • Added hyperlinks to referenced documents	Updated for Quartus II software version 7.2.
May 2007 v7.1.0	• Changed chapter name to Design Recommendations for Altera Devices and the Quartus II Design Assistant • Removed Hierarchical Design Partitioning section • Updated Design Assistant Rules on page 5–19 • Added Finite State Machine Rules on page 5–36 • Added Enabling and Disabling Design Assistant Rules on page 5–38 • Added Rule Suppression Assignments Report on page 5–45 • Added Ignored Design Assistant Assignments Report on page 5–45 • Updated Table 5–2 • Added Referenced Documents on page 5–47	Updated for Quartus II software version 7.1.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Added the following sections (with additional subsections): • "Checking Design Violations Using the Design Assistant" • "Quartus II Design Flow with the Design Assistant" • "The Design Assistant Page" • "Message Severity Levels" • "Design Assistant Rules" • "Viewing Design Assistant Results"	Quartus II software version 6.1 added the Design Assistant; the bulk of the changes to this chapter are related to this update.
May 2006 v6.0.0	Minor updates for the Quartus II version 6.0.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—

Table 5–3. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
May 2005 v.5.0.0	Chapter 5 was formerly Chapter 4 in version 4.2.	—
December 2004 v2.1	Updated for Quartus II software version 4.2: • Chapter 5 was formerly Chapter 6 in version 4.1. • General formatting and editing updates. • Updated hardware requirements for the Quartus II Timing Analyzer. • Added timing requirements and analysis details. • Updated Design Guidelines. • Added information about performing timing analysis on asynchronous ports. • Added inferred latches information. • Updated Delay Chains description. • Updated figures, tables. • Added Clocking Schemes information. • Added details to Multiplexed Clocks details. • Added clock gating details. • Updated Hierarchical Design Partitioning to include synthesis and incremental synthesis. • Added global routing information.	—
June 2004 v.2.0	• Updates to tables, figures, coding examples. • New functionality for Quartus II software 4.1.	—
February 2004 v1.0	Initial release.	—

6. Recommended HDL Coding Styles

QII51007-7.2.0

Introduction

HDL coding styles can have a significant effect on the quality of results that you achieve for programmable logic designs. Synthesis tools optimize HDL code for both logic utilization and performance. However, sometimes the best optimizations require human understanding of the design, and synthesis tools have no information about the purpose or intent of the design. You are often in the best position to improve your quality of results.

This chapter addresses HDL coding style recommendations to ensure optimal synthesis results when targeting Altera® devices, including the following sections:

- "Quartus II Language Templates" on page 6–2
- "Using Altera Megafunctions" on page 6–3
- "Instantiating Altera Megafunctions in HDL Code" on page 6–4
- "Inferring Multiplier and DSP Functions from HDL Code" on page 6–7
- "Inferring Memory Functions from HDL Code" on page 6–13
- "Coding Guidelines for Registers and Latches" on page 6–37
- "General Coding Guidelines" on page 6–48
- "Designing with Low-Level Primitives" on page 6–73



For additional guidelines on structuring your design, refer to the *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*.

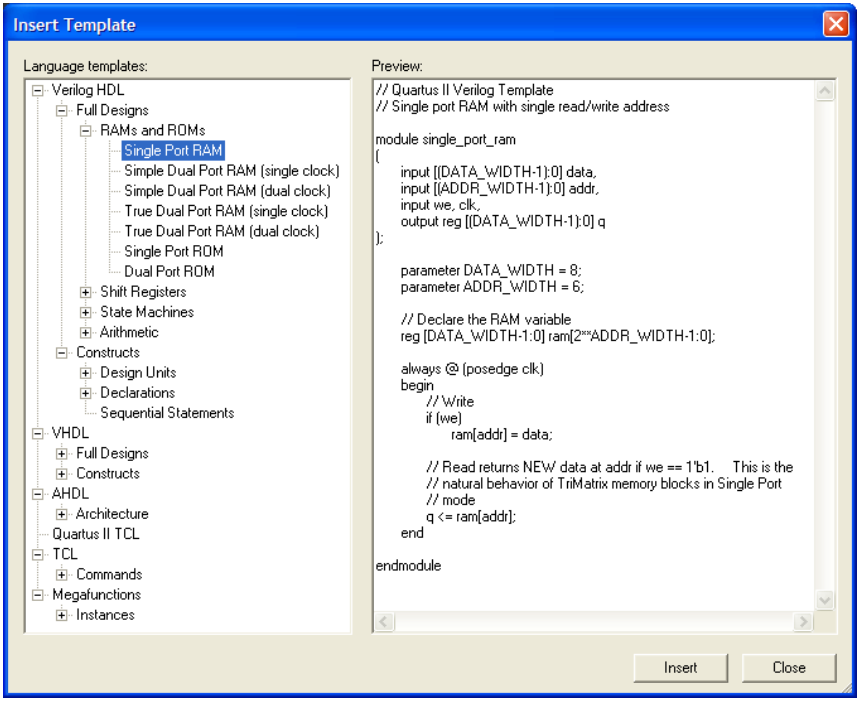
For style recommendations, options, or HDL attributes specific to your synthesis tool (including Quartus® II Integrated Synthesis and other EDA tools), refer to the tool vendor's documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Quartus II Language Templates

The Quartus II software provides Verilog HDL, VHDL, AHDL, Tcl script, and megafunction language templates that can help you with your design.

Many of the Verilog HDL and VHDL examples in this document correspond with examples in the templates. You can easily insert examples from this document into your HDL source code using the **Insert Template** dialog box in the Quartus II user interface, shown in Figure 6–1.

Figure 6–1. Insert Template Dialog Box



To open the Insert Template dialog box when you have a file open in the Quartus II Text Editor, on the Edit menu, click **Insert Template**. Alternately, you can right-click in the Text Editor window and choose **Insert Template**.

Using Altera Megafunctions

Altera provides parameterizable megafunctions that are optimized for Altera device architectures. Using megafunctions instead of coding your own logic saves valuable design time. Additionally, the Altera-provided megafunctions may offer more efficient logic synthesis and device implementation. You can scale the megafunction's size and set various options by setting parameters. Megafunctions include the library of parameterized modules (LPM) and Altera device-specific megafunctions.

To use megafunctions in your HDL code, you can instantiate them as described in "Instantiating Altera Megafunctions in HDL Code" on page 6–4.

Sometimes it is preferable to make your code independent of device family or vendor, and you do not want to instantiate megafunctions directly. For some types of logic functions, such as memory and DSP functions, you can infer a megafunction instead of instantiating it. Synthesis tools, including Quartus II integrated synthesis, recognize certain types of HDL code and automatically infer the appropriate megafunction. The synthesis tool uses the Altera megafunction code when compiling your design—even when you do not specifically instantiate the megafunction. Synthesis tools infer megafunctions to take advantage of logic that is optimized for Altera devices or to target dedicated architectural blocks.

In cases where you prefer to use generic HDL code instead of instantiating a megafunction, follow the guidelines and coding examples in "Inferring Multiplier and DSP Functions from HDL Code" on page 6–7 and "Inferring Memory Functions from HDL Code" on page 6–13 to ensure your HDL code infers the appropriate Altera megafunction.



You must use megafunctions to access some Altera device-specific architecture features. You can infer or instantiate megafunctions to target some features such as memory and DSP blocks. You must instantiate megafunctions to target certain device and high-speed features such as LVDS drivers, PLLs, transceivers, and double-data rate input/output (DDIO) circuitry.

For some designs, generic HDL code can provide better results than instantiating a megafunction. Refer to the following general guidelines and examples that describe when to use standard HDL code and when to use megafunctions:

- For simple addition or subtraction functions, use the + or – symbol instead of an LPM function. Instantiating an LPM function for simple arithmetic operations can result in a less efficient result because the function is hard coded and the synthesis algorithms cannot take advantage of basic logic optimizations.
- For simple multiplexers and decoders, use array notation (such as `out = data[sel]`) instead of an LPM function. Array notation works very well and has simple syntax. You can use the `lpm_mux` function to take advantage of architectural features such as cascade chains in APEX™ series devices, but use the LPM function only if you understand the device architecture in detail and want to force a specific implementation.
- Avoid division operations where possible. Division is an inherently slow operation. Many designers use multiplication creatively to produce division results.

Instantiating Altera Megafunctions in HDL Code

The following sections describe how to use megafunctions by instantiating them in your HDL code with the following methods:

- "Instantiating Megafunctions Using the MegaWizard Plug-In Manager"—You can use the MegaWizard® Plug-In Manager to parameterize the function and create a wrapper file.
- "Creating a Netlist File for Other Synthesis Tools"—You can optionally create a netlist file instead of a wrapper file.
- "Instantiating Megafunctions Using the Port and Parameter Definition"—You can instantiate the function directly in your HDL code.

Instantiating Megafunctions Using the MegaWizard Plug-In Manager

Use the MegaWizard Plug-In Manager as described in this section to create megafunctions in the Quartus II GUI that you can instantiate in your HDL code. The MegaWizard Plug-In Manager provides a graphical user interface to customize and parameterize megafunctions, and ensures that you set all megafunction parameters properly. When you finish setting parameters, you can specify which files you want to be generated. Depending on which language you choose, the MegaWizard Plug-In

Manager instantiates the megafunction with the correct parameters and generates a megafunction variation file (wrapper file) in Verilog HDL (**.v**), VHDL (**.vhd**), or AHDL (**.tdf**) along with other supporting files.

The MegaWizard Plug-In Manager provides options to create the following files:

- A sample instantiation template for the language of the variation file (**_inst.v** | **vhd** | **tdf**).
- Component Declaration File (**.cmp**) that can be used in VHDL Design Files
- ADHL Include File (**.inc**) that can be used in Text Design Files (**.tdf**)
- Quartus II Block Symbol File (**.bsf**) for schematic designs
- Verilog HDL module declaration file that can be used when instantiating the megafunction as a black box in a third-party synthesis tool (**_bb.v**).
- If you enable the option to generate a synthesis area and timing estimation netlist, the MegaWizard Plug-In Manager generates an additional synthesis netlist file (**_syn.v**). Refer to "Creating a Netlist File for Other Synthesis Tools" on page 6–6 for details.

Refer to Table 6–1 for a list and description of files generated by the MegaWizard Plug-In Manager.

Table 6–1. MegaWizard Plug-In Manager Generated Files (Part 1 of 2)

File	Description
<*output file*>**.v** *(1)*	Verilog HDL Variation Wrapper File—Megafunction wrapper file for instantiation in a Verilog HDL design.
<*output file*>**.vhd** *(1)*	VHDL Variation Wrapper File—Megafunction wrapper file for instantiation in a VHDL design.
<*output file*>**.tdf** *(1)*	AHDL Variation Wrapper File—Megafunction wrapper file for instantiation in an AHDL design.
<*output file*>**.inc**	ADHL Include File—Used in AHDL designs.
<*output file*>**.cmp**	Component Declaration File—Used in VHDL designs.
<*output file*>**.bsf**	Block Symbol File—Used in Quartus II Block Design Files (**.bdf**).
<*output file*>**_inst.v**	Verilog HDL Instantiation Template—Sample Verilog HDL instantiation of the module in the megafunction wrapper file.
<*output file*>**_inst.vhd**	VHDL Instantiation Template—Sample VHDL instantiation of the entity in the megafunction wrapper file.
<*output file*>**_inst.tdf**	Text Design File Instantiation Template—Sample AHDL instantiation of the subdesign in the megafunction wrapper file.

Table 6–1. MegaWizard Plug-In Manager Generated Files (Part 2 of 2)

File	Description
<*output file*>**_bb.v**	Black box Verilog HDL Module Declaration—Hollow-body module declaration that can be used in Verilog HDL designs to specify port directions when creating black boxes in third-party synthesis tools.
<*output file*>**_syn.v** *(2)*	Synthesis area and timing estimation netlist—Megafunction netlist used by certain third-party synthesis tools to improve area and timing estimations.

Notes to Table 6–1:

(1) The MegaWizard Plug-In Manager generates either the Verilog HDL, VHDL, or AHDL Variation Wrapper File, depending on the language you select for the output file on the megafunction-selection page of the wizard.

(2) The MegaWizard Plug-In Manager generates this file only if you turn on the **Generate a synthesis area and timing estimation netlist** option on the EDA page of the wizard.

Creating a Netlist File for Other Synthesis Tools

When you use certain megafunctions with third-party EDA synthesis tools (that is, tools other than Quartus II integrated synthesis), you can optionally create a netlist for area and timing estimation instead of a wrapper file.

The netlist file is a representation of the customized logic used in the Quartus II software. The file provides the connectivity of architectural elements in the megafunction but may not represent true functionality. This information enables certain third-party synthesis tools to better report area and timing estimates. In addition, synthesis tools can use the timing information to focus timing-driven optimizations and improve the quality of results.



For information about support for area and timing estimation netlists in your synthesis tool, refer to the tool vendor's documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

To generate the netlist, turn on **Generate a synthesis area and timing estimation netlist** on the EDA page of the MegaWizard Plug-In Manager. The netlist file is called <*output file*>**_syn.v**.

Instantiating Megafunctions Using the Port and Parameter Definition

You can instantiate the megafunction directly in your Verilog HDL, VHDL, or AHDL code by calling the megafunction and setting its parameters as you would any other module, component, or subdesign.



Refer to the specific megafunction in the Quartus II Help for a list of the megafunction ports and parameters. Quartus II Help also provides a sample VHDL component declaration and AHDL function prototype for each megafunction.



Altera strongly recommends that you use the MegaWizard Plug-In Manager for complex megafunctions such as PLLs, transceivers, and LVDS drivers. For details on using the MegaWizard Plug-In Manager, refer to "Instantiating Megafunctions Using the MegaWizard Plug-In Manager" on page 6–4.

Inferring Multiplier and DSP Functions from HDL Code

The following sections describe how to infer multiplier and DSP functions from generic HDL code, and, if applicable, how to target the dedicated DSP block architecture in Altera devices:

- "Multipliers—Inferring the lpm_mult Megafunction from HDL Code" on page 6–7
- "Multiply-Accumulators and Multiply-Adders—Inferring altmult_accum and altmult_add Megafunctions from HDL Code" on page 6–10



For synthesis tool features and options, refer to your synthesis tool documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Multipliers—Inferring the lpm_mult Megafunction from HDL Code

To infer multiplier functions, synthesis tools look for multipliers and convert them to `lpm_mult` or `altmult_add` megafunctions, or may map them directly to device atoms. For devices with DSP blocks, the software can implement the function in a DSP block instead of logic,

depending on device utilization. The Quartus II Fitter can also place input and output registers in DSP blocks (that is, perform register packing) to improve performance and area utilization.

For additional information about the DSP block and the supported functions, refer to the appropriate Altera device family handbook and Altera's DSP Solutions Center website at **www.altera.com**.

The following four code samples show Verilog HDL and VHDL examples for unsigned and signed multipliers that synthesis tools can infer as an `lpm_mult` or `altmult_add` megafunction. Each example fits into one DSP block 9-bit element. In addition, when register packing occurs, no extra logic cells for registers are required.

The `signed` declaration in Verilog HDL is a feature of the Verilog 2001 Standard.

Example 6–1. Verilog HDL Unsigned Multiplier

```
module unsigned_mult (out, a, b);
   output [15:0] out;
   input  [7:0] a;
   input  [7:0] b;
   assign out = a * b;
endmodule
```

Example 6–2. Verilog HDL Signed Multiplier with Input and Output Registers (Pipelining = 2)

```
module signed_mult (out, clk, a, b);
   output [15:0] out;
   input clk;
   input signed [7:0] a;
   input signed [7:0] b;

   reg signed [7:0] a_reg;
   reg signed [7:0] b_reg;
   reg signed [15:0] out;
   wire signed [15:0] mult_out;

   assign mult_out = a_reg * b_reg;

   always @ (posedge clk)
   begin
      a_reg <= a;
      b_reg <= b;
      out <= mult_out;
   end
endmodule
```

Example 6–3. VHDL Unsigned Multiplier with Input and Output Registers (Pipelining = 2)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY unsigned_mult IS
   PORT (
      a: IN UNSIGNED (7 DOWNTO 0);
      b: IN UNSIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      result: OUT UNSIGNED (15 DOWNTO 0)
   );
END unsigned_mult;

ARCHITECTURE rtl OF unsigned_mult IS
   SIGNAL a_reg, b_reg: UNSIGNED (7 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr ='1') THEN
         a_reg <= (OTHERS => '0');
         b_reg <= (OTHERS => '0');
         result <= (OTHERS => '0');
      ELSIF (clk'event AND clk = '1') THEN
         a_reg <= a;
         b_reg <= b;
         result <= a_reg * b_reg;
      END IF;
   END PROCESS;
END rtl;
```

Example 6–4. VHDL Signed Multiplier

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY signed_mult IS
   PORT (
      a: IN SIGNED (7 DOWNTO 0);
      b: IN SIGNED (7 DOWNTO 0);
      result: OUT SIGNED (15 DOWNTO 0)
   );
END signed_mult;

BEGIN
   result <= a * b;
END rtl;
```

Multiply-Accumulators and Multiply-Adders—Inferring altmult_accum and altmult_add Megafunctions from HDL Code

Synthesis tools detect multiply-accumulators or multiply-adders and convert them to altmult_accum or altmult_add megafunctions, respectively, or may map them directly to device atoms. The Quartus II software then places these functions in DSP blocks during placement and routing.



Synthesis tools infer multiply-accumulator and multiply-adder functions only if the Altera device family has dedicated DSP blocks that support these functions.

A multiply-accumulator consists of a multiplier feeding an addition operator. The addition operator feeds a set of registers that then feeds the second input to the addition operator. A multiply-adder consists of two to four multipliers feeding one or two levels of addition, subtraction, or addition/subtraction operators. Addition is always the second-level operator, if it is used. In addition to the multiply-accumulator and multiply-adder, the Quartus II Fitter also places input and output registers into the DSP blocks to pack registers and improve performance and area utilization.

The Verilog HDL and VHDL code samples shown in Examples 6–5 through 6–8 infer specific multiply-accumulators and multiply-adders.

Example 6–5. Verilog HDL Unsigned Multiply-Accumulator with Input, Output and Pipeline Registers (Latency = 3)

```
module unsig_altmult_accum (dataout, dataa, datab, clk, aclr, clken);
   input [7:0] dataa;
   input [7:0] datab;
   input clk;
   input aclr;
   input clken;
   output [31:0] dataout;
   reg [31:0] dataout;
   reg [7:0] dataa_reg;
   reg [7:0] datab_reg;
   reg [15:0] multa_reg;
   wire [15:0] multa;
   wire [31:0] adder_out;
   assign multa = dataa_reg * datab_reg;
   assign adder_out = multa_reg + dataout;
   always @ (posedge clk or posedge aclr)
   begin
      if (aclr)
      begin
         dataa_reg <= 8'b0;
         datab_reg <= 8'b0;
         multa_reg <= 16'b0;
         dataout <= 32'b0;
      end
      else if (clken)
      begin
         dataa_reg <= dataa;
         datab_reg <= datab;
         multa_reg <= multa;
         dataout <= adder_out;
      end
   end
endmodule
```

Example 6–6. Verilog HDL Signed Multiply-Adder (Latency = 0)

```
module sig_altmult_add (dataa, datab, datac, datad, result);
   input signed [15:0] dataa;
   input signed [15:0] datab;
   input signed [15:0] datac;
   input signed [15:0] datad;
   output [32:0] result;

   wire signed [31:0] mult0_result;
   wire signed [31:0] mult1_result;

   assign mult0_result = dataa * datab;
   assign mult1_result = datac * datad;
   assign result = (mult0_result + mult1_result);
endmodule
```

Example 6–7. VHDL Unsigned Multiply-Adder with Input, Output and Pipeline Registers (Latency = 3)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY unsignedmult_add IS
   PORT (
      a: IN UNSIGNED (7 DOWNTO 0);
      b: IN UNSIGNED (7 DOWNTO 0);
      c: IN UNSIGNED (7 DOWNTO 0);
      d: IN UNSIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      result: OUT UNSIGNED (15 DOWNTO 0)
   );
END unsignedmult_add;

ARCHITECTURE rtl OF unsignedmult_add IS
   SIGNAL a_reg, b_reg, c_reg, d_reg: UNSIGNED (7 DOWNTO 0);
   SIGNAL pdt_reg, pdt2_reg: UNSIGNED (15 DOWNTO 0);
   SIGNAL result_reg: UNSIGNED (15 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr = '1') THEN
         a_reg <= (OTHERS => '0');
         b_reg <= (OTHERS => '0');
         c_reg <= (OTHERS => '0');
         d_reg <= (OTHERS => '0');
         pdt_reg <= (OTHERS => '0');
         pdt2_reg <= (OTHERS => '0');

      ELSIF (clk'event AND clk = '1') THEN
         a_reg <= a;
         b_reg <= b;
         c_reg <= c;
         d_reg <= d;
         pdt_reg <= a_reg * b_reg;
         pdt2_reg <= c_reg * d_reg;
         result_reg <= pdt_reg + pdt2_reg;
      END IF;
      END PROCESS;
   result <= result_reg;
END rtl;
```

Example 6–8. VHDL Signed Multiply-Accumulator with Input, Output and Pipeline Registers (Latency = 3)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY sig_altmult_accum IS
   PORT (
      a: IN SIGNED(7 DOWNTO 0);
      b: IN SIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      accum_out: OUT SIGNED (15 DOWNTO 0)
   ) ;
END sig_altmult_accum;

ARCHITECTURE rtl OF sig_altmult_accum IS
   SIGNAL a_reg, b_reg: SIGNED (7 DOWNTO 0);
   SIGNAL pdt_reg: SIGNED (15 DOWNTO 0);
   SIGNAL adder_out: SIGNED (15 DOWNTO 0);
BEGIN
   PROCESS (clk)
   BEGIN
      IF (clk'event and clk = '1') THEN
         a_reg <= (a);
         b_reg <= (b);
            pdt_reg <= a_reg * b_reg;
         adder_out <= adder_out + pdt_reg;
      END IF;
   END process;
   accum_out <= adder_out;
END rtl;
```

Inferring Memory Functions from HDL Code

The following sections describe how to infer memory functions from generic HDL code and, if applicable, to target the dedicated memory architecture in Altera devices:

- "RAM Functions—Inferring altsyncram and altdpram Megafunctions from HDL Code" on page 6–14
- "ROM Functions—Inferring altsyncram and lpm_rom Megafunctions from HDL Code" on page 6–31
- "Shift Registers—Inferring the altshift_taps Megafunction from HDL Code" on page 6–33



For synthesis tool features and options, refer to your synthesis tool documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Altera's dedicated memory architecture offers a number of advanced features that can be easily targeted using the MegaWizard Plug-In Manager as described in "Instantiating Altera Megafunctions in HDL Code" on page 6–4. The coding recommendations in the following

sections provide portable examples of generic HDL code that infer the appropriate megafunction. However, if you want to use some of the advanced memory features in Altera devices, consider using the megafunction directly so that you can control the ports and parameters more easily.

RAM Functions—Inferring altsyncram and altdpram Megafunctions from HDL Code

To infer RAM functions, synthesis tools detect sets of registers and logic that can be replaced with the `altsyncram` or `altdpram` megafunctions for device families that have dedicated RAM blocks, or may map them directly to device memory atoms.

Standard synthesis tools recognize single-port and simple dual-port (one read port and one write port) RAM blocks. Some tools (such as the Quartus II software) also recognize true dual port RAM blocks that map to the memory blocks in certain Altera devices. Tools usually do not infer small RAM blocks because small RAM blocks typically can be implemented more efficiently using the registers in regular logic.



If you are using Quartus II integrated synthesis, you can direct the software to infer ROM blocks for all sizes with the **Allow Any RAM Size for Recognition** option under **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box.



If your design contains a RAM block that your synthesis tool does not recognize and infer, the design might require a large amount of system memory that potentially can cause compilation problems.

Some synthesis tools provide options to control the implementation of inferred RAM blocks for Altera devices with TriMatrix™ memory blocks. For example, Quartus II integrated synthesis provides the `ramstyle` synthesis attribute to specify the type of memory block or to specify the use of regular logic instead of a dedicated memory block. Quartus II integrated synthesis does not map inferred memory into Stratix III MLABs unless the HDL code specifies the appropriate `ramstyle` attribute, although the Fitter may map some memories to MLABs.



For details about using the `ramstyle` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

When you are using a formal verification flow, Altera recommends that you create RAM blocks in separate entities or modules that contain only the RAM logic. In certain formal verification flows, for example, when using Quartus II integrated synthesis, the entity or module containing the inferred RAM is put into a black box automatically because formal verification tools do not support RAM blocks. The Quartus II software issues a warning message when this occurs. If the entity or module contains any additional logic outside the RAM block, this logic also must be treated as a black box for formal verification and therefore cannot be verified.

This section presents several guidelines for inferring RAM functions that match the dedicated memory architecture in Altera devices, and then provides recommended HDL code for different types of memory logic.

Use Synchronous Memory Blocks

Altera recommends using synchronous memory blocks for Altera designs. The TriMatrix memory blocks in Altera's newest devices are synchronous, so RAM designs that are targeted towards architectures that contain these dedicated memory blocks must be synchronous to be mapped directly into the device architecture. Asynchronous memory logic is not inferred as a memory block or placed in the device dedicated memory blocks; the logic is implemented in regular logic cells.

Synchronous memories are supported in all Altera device families. A memory block is considered synchronous if it uses one of the following read behaviors:

- Memory read occurs in a Verilog always block with a clock signal or a VHDL clocked process.
- Memory read occurs outside a clocked block, but there is a synchronous read address (that is, the address used in the read statement is registered). This type of logic is not always inferred as a memory block, depending on the target device architecture.



The synchronous memory structures in Altera devices differ from the structures in other vendors' devices. Match your design to the target device architecture to achieve the best results.

Later subsections provide coding recommendations for various memory types. All of these examples are synchronous to ensure that they can be directly mapped into the dedicated memory architecture available in Altera FPGAs.



For additional information about the dedicated memory blocks in your specific device, refer to the appropriate Altera device family data sheet on the Altera website at **www.altera.com**.

Avoid Unsupported Reset Conditions

You cannot clear the RAM contents of Altera memory blocks. If your HDL code describes a RAM with a reset signal for the RAM contents, the logic is not inferred as a memory block or mapped to dedicated memory architecture. As a general rule, avoid putting RAM read or write operations in an always block or process block with a reset signal.

Example 6–9 shows an example of undesirable code where there is a reset signal that clears part of the RAM contents. Avoid this coding style because it is not supported in Altera memories.

Example 6–9. Verilog RAM with Reset Signal that Clears RAM Contents: Not Supported in Device Architecture

```
module clear_ram
(
    input clock,
    input reset,
    input we,
    input [7:0] data_in,
    input [4:0] address,
    output reg [7:0] data_out
);

    reg [7:0] mem [0:31];
    integer i;

    always @ (posedge clock or posedge reset)
    begin
        if (reset == 1'b1)
            mem[address] <= 0;
        else if (we == 1'b1)
            mem[address] <= data_in;

        data_out <= mem[address];
    end
endmodule
```

Example 6–10 shows an example of undesirable code where the reset signal affects the RAM, although the effect may not be intended. Avoid this coding style because it is not supported in Altera memories.

Example 6–10. Verilog RAM with Reset Signal that Affects RAM: Not Supported in Device Architecture

```
module bad_reset
(
    input clock,
    input reset,
    input we,
    input [7:0] data_in,
    input [4:0] address,
    output reg [7:0] data_out,
    input d,
    output reg q
);

    reg [7:0] mem [0:31];
    integer i;

    always @ (posedge clock or posedge reset)
    begin
        if (reset == 1'b1)
            q <= 0;
        else
        begin
            if (we == 1'b1)
                mem[address] <= data_in;

            data_out <= mem[address];
            q <= d;
        end
    end
endmodule
```

Check Read-During-Write Behavior

It is important to check the read-during-write behavior of the memory block described in your HDL design as compared to the behavior in your target device architecture. HDL source code specifies the memory behavior when you attempt to read and write from the same memory address in the same clock cycle. The code specifies that the read returns either the old data at the address, or the new data being written to the address. This is referred to as the read-during-write behavior of the memory block. Altera memory blocks have different read-during-write behavior depending on the target device family, memory mode and block type.

Synthesis tools map an HDL design into the target device architecture, with the goal of maintaining the functionality described in your source code. In some cases, memory blocks map directly into the device

architecture; however, in some cases, the device architecture cannot implement the memory behavior described in your source code, so the logic is not mapped to the dedicated memory blocks in the device. In still other cases, the software can implement the memory functionality using some extra logic in addition to the dedicated RAM block. To implement the behavior in the target device, synthesis software may add bypass logic around the memory block, which increases the area utilization of the design and decreases the performance if the memory block is part of the design's critical path.

In many synthesis tools, you can specify that the read-during-write behavior is not important to your design; for example, if you never read from the same address to which you write in the same clock cycle. For Quartus II integrated synthesis, add the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than use the behavior specified by your HDL code. Using this type of attribute prevents the synthesis tool from using extra logic to implement the memory block, and in some cases, can allow memory inference when it would otherwise be impossible.



For more information about attribute syntax, the `no_rw_check` attribute value, or specific options for your synthesis tool, refer to your synthesis tool documentation or to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

The following subsections provide coding recommendations for various memory types. Each example describes the read-during-write behavior and addresses the support for the memory type in Altera devices.

Single-Clock Synchronous RAM with Old Data Read-During-Write Behavior

The code examples in this section show Verilog HDL and VHDL code that infers simple dual port, single-clock synchronous RAM. Single-port RAM blocks use a similar coding style.

The read-during-write behavior in these examples is to read the old data at the memory address. Refer to "Check Read-During-Write Behavior" on page 6–17 for details. Altera recommends that you use this coding style as long as your design does not require that a simultaneous read and write to the same RAM location read the new value that is currently being written to that RAM location.

If you require that the read-during-write results in new data, refer to "Single-Clock Synchronous RAM with New Data Read-During-Write Behavior" on page 6–20.

The simple dual-port RAM code samples shown in Examples 6–11 and 6–12 map directly into Altera TriMatrix memory.

Single-port versions of memory blocks (that is, using the same read address and write address signals) can allow better RAM utilization than dual-port memory blocks, depending on the device family.

Example 6–11. Verilog HDL Single-Clock Simple Dual-Port Synchronous RAM with Old Data Read-During-Write Behavior

```
module single_clk_ram(
    output reg [7:0] q,
    input [7:0] d,
    input [6:0] write_address, read_address,
    input we, clk
);
    reg [7:0] mem [127:0];

    always @ (posedge clk) begin
        if (we)
            mem[write_address] <= d;
        q <= mem[read_address]; // q doesn't get d in this clock cycle
    end
endmodule
```

Example 6–12. VHDL Single-Clock Simple Dual-Port Synchronous RAM with Old Data Read-During-Write Behavior

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY single_clock_ram IS
    PORT (
        clock: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END single_clock_ram;

ARCHITECTURE rtl OF single_clock_ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ram_block: MEM;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
            q <= ram_block(read_address);
            -- VHDL semantics imply that q doesn't get data
            -- in this clock cycle
        END IF;
    END PROCESS;
END rtl;
```

Single-Clock Synchronous RAM with New Data Read-During-Write Behavior

These examples describe RAM blocks in which a simultaneous read and write to the same location reads the new value that is currently being written to that RAM location.

To implement this behavior in the target device, synthesis software adds bypass logic around the RAM block. This bypass logic increases the area utilization of the design and decreases the performance if the RAM block is part of the design's critical path. Refer to "Check Read-During-Write Behavior" on page 6–17 for details. If this behavior is not required for your design, use the examples from "Single-Clock Synchronous RAM with Old Data Read-During-Write Behavior" on page 6–18.

The simple dual-port RAM examples shown in Examples 6–13 and 6–14 require bypass the software to create this logic around the RAM block.

Single-port versions of the Verilog memory block (that is, using the same read address and write address signals) do not require any logic cells to create bypass logic in Arria™ GX devices, and Stratix® and Cyclone® series of devices, because the device memory supports new data read-during-write behavior when in single-port mode (same clock, same read and write address).

Example 6–13. Verilog HDL Single-Clock Simple Dual-Port Synchronous RAM with New Data Read-During-Write Behavior

```
module single_clock_wr_ram(
    output reg [7:0] q,
    input [7:0] d,
    input [6:0] write_address, read_address,
    input we, clk
);
    reg [7:0] mem [127:0];

    always @ (posedge clk) begin
        if (we)
            mem[write_address] = d;
        q = mem[read_address]; // q does get d in this clock cycle if we is high
    end
endmodule
```

Note that Example 6–13 is similar to Example 6–11, but Example 6–13 uses a blocking assignment for the write so that the data is assigned immediately.

An alternative way to create a single-clock RAM is to use an assign statement to read the address of `mem` to create the output `q`, as shown in following the coding style. By itself, the code describes new data read-during-write behavior. However, if the RAM output feeds a register in another hierarchy, then a read-during-write would result in the old data. Synthesis tools may not infer a RAM block if the tool cannot determine which behavior is described, such as when the memory feeds a hard hierarchical partition boundary. For this reason, avoid using this alternate type of coding style.

```
reg [7:0] mem [127:0];
reg [6:0] read_address_reg;

always @ (posedge clk) begin
    if (we)
        mem[write_address] <= d;

    read_address_reg <= read_address;
end

assign q = mem[read_address_reg];
```

Example 6–14 uses a concurrent signal assignment to read from the RAM. By itself, this example describes new data read-during-write behavior. However, if the RAM output feeds a register in another hierarchy, then a read-during-write would result in the old data. Synthesis tools may not infer a RAM block if the tool cannot determine which behavior is described, such as when the memory feeds a hard hierarchical partition boundary.

Example 6–14. VHDL Single-Clock Simple Dual-Port Synchronous RAM with New Data Read-During-Write Behavior

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY single_clock_rw_ram IS
    PORT (
        clock: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END single_clock_rw_ram;

ARCHITECTURE rtl OF single_clock_rw_ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ram_block: MEM;
    SIGNAL read_address_reg: INTEGER RANGE 0 to 31;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
            read_address_reg <= read_address;
        END IF;
    END PROCESS;
    q <= ram_block(read_address_reg);
END rtl;
```

This example does not infer a RAM block for APEX, ACEX, or FLEX devices by default because the read-during-write behavior depends on surrounding logic. For Quartus II integrated synthesis, if you do not require the read-through-write capability, add the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than use the behavior specified by your HDL code.

Simple Dual-Port, Dual-Clock Synchronous RAM

In dual clock designs, synthesis tools cannot accurately infer the read-during-write behavior because it depends on the timing of the two clocks within the target device. Therefore, the read-during-write behavior of the synthesized design is undefined and may differ from your original HDL code. Refer to "Check Read-During-Write Behavior" on page 6–17 for details.

When Quartus II integrated synthesis infers this type of RAM, it issues a warning because of the undefined read-during-write behavior. If this functionality is acceptable in your design, you can avoid the warning by adding the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM.

The code samples shown in Examples 6–15 and 6–16 show Verilog HDL and VHDL code that infers dual-clock synchronous RAM. The exact behavior depends on the relationship between the clocks.

Example 6–15. Verilog HDL Simple Dual-Port, Dual-Clock Synchronous RAM

```
module dual_clock_ram(
   output reg [7:0] q,
   input [7:0] d,
   input [6:0] write_address, read_address,
   input we, clk1, clk2
);
   reg [6:0] read_address_reg;
   reg [7:0] mem [127:0];

   always @ (posedge clk1)
   begin
      if (we)
         mem[write_address] <= d;
   end

   always @ (posedge clk2) begin
      q <= mem[read_address_reg];
      read_address_reg <= read_address;
   end
endmodule
```

Example 6–16. VHDL Simple Dual-Port, Dual-Clock Synchronous RAM

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY dual_clock_ram IS
    PORT (
        clock1, clock2: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
```

```
    );
END dual_clock_ram;
ARCHITECTURE rtl OF dual_clock_ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL ram_block: MEM;
    SIGNAL read_address_reg : INTEGER RANGE 0 to 31;
BEGIN
    PROCESS (clock1)
    BEGIN
        IF (clock1'event AND clock1 = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
        END IF;
    END PROCESS;
    PROCESS (clock2)
    BEGIN
        IF (clock2'event AND clock2 = '1') THEN
            q <= ram_block(read_address_reg);
            read_address_reg <= read_address;
        END IF;
    END PROCESS;
END rtl;
```

True Dual-Port Synchronous RAM

The code examples in this section show Verilog HDL and VHDL code that infers true dual-port synchronous RAM. Different synthesis tools may differ in their support for these types of memories. This section describes the inference rules for Quartus II integrated synthesis. This type of RAM inference is supported only for Arria GX devices, and the Stratix and Cyclone series of devices.

Altera TriMatrix memory blocks have two independent address ports, allowing for operations on two unique addresses simultaneously. A read operation and a write operation can share the same port if they share the same address. The Quartus II software infers true dual-port RAMs in Verilog HDL and VHDL with any combination of independent read or write operations in the same clock cycle, with at most two unique port addresses, performing two reads and one write, two writes and one read, or two writes and two reads in one clock cycle with one or two unique addresses.

In the TriMatrix RAM block architecture, there is no priority between the two ports. Therefore if you write to the same location on both ports at the same time, the result is indeterminate. You must ensure your HDL code does not imply priority for writes to the memory block. For example, if both ports are defined in the same process block, the code is synthesized

and simulated sequentially so there would be a priority between the two ports. If you code does imply a priority, the logic cannot be implemented in the device RAM blocks.

You must also consider the read-during-write behavior of the RAM block, to ensure that it can be mapped directly to the device RAM architecture. Refer to "Check Read-During-Write Behavior" on page 6–17 for details.

When a read and write operation occur on the same port for the same address, the read operation may behave as follows:

- **Read new data.** This mode matches the behavior of TriMatrix memory blocks.
- **Read old data.** This mode is supported only by Stratix III and Cyclone III TriMatrix memory blocks. This behavior is not possible in TriMatrix memory blocks of other families.

When a read and write operation occur on different ports for the same address (also known as mixed port), the read operation may behave as follows:

- **Read new data.** Quartus II integrated synthesis supports this mode by creating bypass logic around the TriMatrix memory block.
- **Read old data.** This behavior is supported by TriMatrix memory blocks.

The Verilog HDL single-clock code sample shown in Example 6–17 maps directly into Altera TriMatrix memory. When a read and write operation occur on the same port for the same address, the new data being written to the memory is read. When a read and write operation occur on different ports for the same address, the old data in the memory is read. Simultaneous writes to the same location on both ports results in indeterminate behavior.

A dual-clock version of this design describes the same behavior, but the memory in the target device will have undefined mixed port read-during-write behavior because it depends on the relationship between the clocks.

Example 6–17. Verilog HDL True Dual-Port RAM with Single Clock

```
module true_dual_port_ram_single_clock
(
    input [(DATA_WIDTH-1):0] data_a, data_b,
    input [(ADDR_WIDTH-1):0] addr_a, addr_b,
    input we_a, we_b, clk,
    output reg [(DATA_WIDTH-1):0] q_a, q_b
);
```

```
    parameter DATA_WIDTH = 8;
    parameter ADDR_WIDTH = 6;

    // Declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

    always @ (posedge clk)
    begin // Port A
        if (we_a)
        begin
            ram[addr_a] <= data_a;
            q_a <= data_a;
        end
        else
            q_a <= ram[addr_a];
    always @ (posedge clk)
    begin // Port b
        if (we_b)
        begin
            ram[addr_b] <= data_b;
            q_b <= data_b;
        end
        else
            q_b <= ram[addr_b];
    end

endmodule
```

If you use Verilog read statements shown below instead of the `if-else` statements in Example 6–17, the read results in old data when a read and write operation occur at the same time for the same address on the same port or mixed ports. This behavior is supported only in the TriMatrix memories of Stratix III and Cyclone III devices, and is not inferred as memory for other device families.

```
always @ (posedge clk)
begin // Port A
  if (we_a)
     ram[addr_a] <= data_a;

  q_a <= ram[addr_a];
end

always @ (posedge clk)
begin // Port B
  if (we_b)
     ram[addr_b] <= data_b;

  q_b <= ram[addr_b];
end
```

The VHDL single-clock code sample shown in Example 6–18 maps directly into Altera TriMatrix memory. When a read and write operation occur on the same port for the same address, the new data being written to the memory is read. When a read and write operation occur on

different ports for the same address, the old data in the memory is read. Simultaneous writes to the same location on both ports results in indeterminate behavior.

A dual-clock version of this design describes the same behavior, but the memory in the target device will have undefined mixed port read-during-write behavior because it depends on the relationship between the clocks.

Example 6–18. VHDL True Dual-Port RAM with Single Clock

```
library ieee;
use ieee.std_logic_1164.all;

entity true_dual_port_ram_single_clock is

    generic
    (
        DATA_WIDTH : natural := 8;
        ADDR_WIDTH : natural := 6
    );

    port
    (
        clk : in std_logic;
        addr_a: in natural range 0 to 2**ADDR_WIDTH - 1;
        addr_b: in natural range 0 to 2**ADDR_WIDTH - 1;
        data_a: in std_logic_vector((DATA_WIDTH-1) downto 0);
        data_b: in std_logic_vector((DATA_WIDTH-1) downto 0);
        we_a: in std_logic := '1';
        we_b: in std_logic := '1';
        q_a : out std_logic_vector((DATA_WIDTH -1) downto 0)
        q_b : out std_logic_vector((DATA_WIDTH -1) downto 0)
    );

end true_dual_port_ram_single_clock;

architecture rtl of true_dual_port_ram_single_clock is

    -- Build a 2-D array type for the RAM
    subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);
    type memory_t is array(raddr'high downto 0) of word_t;

    -- Declare the RAM signal.
    signal ram : memory_t;

begin

    process(clk)
    begin
    if(rising_edge(clk)) then -- Port A
        if(we_a = '1') then
            ram(addr_a) <= data_a;

            -- Read-during-write on the same port returns NEW data
            q_a <= data_a;
        else
            -- Read-during-write on the mixed port returns OLD data
```

```
            q_a <= ram(addr_a);
        end if;

    end process;

    process(clk)
    begin
    if(rising_edge(clk)) then -- Port B
        if(we_b = '1') then
            ram(addr_b) <= data_b;

            -- Read-during-write on the same port returns NEW data
            q_b <= data_b;
        else
            -- Read-during-write on the mixed port returns OLD data
            q_b <= ram(addr_b);
        end if;
    end if;
    end process;

end rtl;
```

Specifying Initial Memory Contents at Power-Up

Your synthesis tool may offer various ways to specify the initial contents of an inferred memory.



Certain device memory types do not support initialized memory, such as the M-RAM blocks in Stratix and Stratix II devices.



Note that there are slight power-up and initialization differences between dedicated RAM blocks and the Stratix III MLAB memory due to the continuous read of the MLAB. Altera dedicated RAM block outputs always power-up to zero and are set to the initial value on the first read. For example, if address 0 is pre-initialized to FF, the RAM block powers up with the output at 0. A subsequent read after power up from address 0 outputs the pre-initialized value of FF. Therefore, if a RAM is powered up and an enable (read enable or clock enable) is held low, then the power-up output of "0" is maintained until the first valid read cycle. The Stratix III MLAB is implemented using registers that power-up to 0, but are initialized to their initial value immediately at power-up or reset. You will therefore see the initial value regardless of the enable status. Quartus II integrated synthesis does not map inferred memory to MLABs unless the HDL code specifies the appropriate ramstyle attribute.

Quartus II integrated synthesis supports the `ram_init_file` synthesis attribute that allows you to specify a Memory Initialization File (**.mif**) for an inferred RAM block.



For information about the `ram_init_file` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the tool vendor's documentation.

In Verilog HDL, you can use an initial block to initialize the contents of an inferred memory. Quartus II integrated synthesis automatically converts the initial block into a MIF for the inferred RAM. Example 6–19 shows Verilog HDL code that infers a simple dual-port RAM block and corresponding MIF file.

Example 6–19. Verilog HDL RAM with Initialized Contents

```
module ram_with_init(
   output reg [7:0] q,
   input [7:0] d,
   input [4:0] write_address, read_address,
   input we, clk
);
   reg [7:0] mem [0:31];
   integer i;

   initial begin
      for (i = 0; i < 32; i = i + 1)
         mem[i] = i[7:0];
   end

   always @ (posedge clk) begin
      if (we)
         mem[write_address] <= d;
      q <= mem[read_address];
   end
endmodule
```

Quartus II integrated synthesis and other synthesis tools also support the `$readmemb` and `$readmemh` commands so that RAM and ROM initialization work identically in synthesis and simulation. Example 6–20 shows an initial block that initializes an inferred RAM block using the `$readmemb` command.

Example 6–20. Verilog HDL RAM Initialized with the readmemb Command

```
reg [7:0] ram[0:15];
initial
begin
  $readmemb("ram.txt", ram);
end
```

In VHDL, you can initialize the contents of an inferred memory by specifying a default value for the corresponding signal. Quartus II integrated synthesis automatically converts the default value into a MIF for the inferred RAM. Example 6–21 shows VHDL code that infers a simple dual-port RAM block and corresponding MIF file.

Example 6–21. VHDL RAM with Initialized Contents

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;

ENTITY ram_with_init IS
    PORT(
            clock: IN STD_LOGIC;
            data: IN UNSIGNED (7 DOWNTO 0);
            write_address: IN integer RANGE 0 to 31;
            read_address: IN integer RANGE 0 to 31;
            we: IN std_logic;
            q: OUT UNSIGNED (7 DOWNTO 0));
END;

ARCHITECTURE rtl OF ram_with_init IS

    TYPE MEM IS ARRAY(31 DOWNTO 0) OF unsigned(7 DOWNTO 0);
    FUNCTION initialize_ram
        return MEM is
        variable result : MEM;
    BEGIN
        FOR i IN 31 DOWNTO 0 LOOP
            result(i) := to_unsigned(natural(i), natural'(8));
        END LOOP;
        RETURN result;
    END initialize_ram;

    SIGNAL ram_block : MEM := initialize_ram;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
            ram_block(write_address) <= data;
            END IF;
            q <= ram_block(read_address);
        END IF;
    END PROCESS;
END rtl;
```

ROM Functions—Inferring altsyncram and lpm_rom Megafunctions from HDL Code

To infer ROM functions, synthesis tools detect sets of registers and logic that can be replaced with the `altsyncram` or `lpm_rom` megafunctions, depending on the target device family, only for device families that have dedicated memory blocks.

ROMs are inferred when a `case` statement exists in which a value is set to a constant for every choice in the case statement. Because small ROMs typically achieve the best performance when they are implemented using the registers in regular logic, each ROM function must meet a minimum size requirement to be inferred and placed into memory.



If you are using Quartus II integrated synthesis, you can direct the software to infer ROM blocks for all sizes with the **Allow Any ROM Size for Recognition** option under **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box.

Some synthesis tools provide options to control the implementation of inferred ROM blocks for Altera devices with TriMatrix memory blocks. For example, Quartus II integrated synthesis provides the `romstyle` synthesis attribute to specify the type of memory block or to specify the use of regular logic instead of a dedicated memory block.



For details about using the `romstyle` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

When you are using a formal verification flow, Altera recommends that you create ROM blocks in separate entities or modules that contain only the ROM logic because you may need to treat the entity and module as a black box during formal verification.



Because formal verification tools do not support ROM megafunctions, Quartus II integrated synthesis does not infer ROM megafunctions when a formal verification tool is selected.

The Verilog HDL and VHDL code samples shown in Examples 6–22 and 6–23 infer synchronous ROM blocks. Depending on the device family's dedicated RAM architecture, the ROM logic may have to be synchronous; consult the device family handbook for details.

For device architectures with synchronous RAM blocks, such as the Stratix series devices and newer device families, either the address or the output has to be registered for ROM code to be inferred. When output registers are used, the registers are implemented using the input registers of the RAM block, but the functionality of the ROM is not changed. If you register the address, the power-up state of the inferred ROM can be different from the HDL design. In this scenario, the synthesis software issues a warning. The Quartus II Help explains the condition under which the functionality changes when you are using Quartus II integrated synthesis.

These ROM code samples map directly to the Altera TriMatrix memory architecture.

Example 6–22. Verilog HDL Synchronous ROM

```
module sync_rom (clock, address, data_out);
    input clock;
    input [7:0] address;
    output [5:0] data_out;

    reg [5:0] data_out;

    always @ (posedge clock)
    begin
        case (address)
            8'b00000000: data_out = 6'b101111;
            8'b00000001: data_out = 6'b110110;
            ...
            8'b11111110: data_out = 6'b000001;
            8'b11111111: data_out = 6'b101010;
        endcase
    end
endmodule
```

Example 6–23. VHDL Synchronous ROM

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY sync_rom IS
    PORT (
        clock: IN STD_LOGIC;
        address: IN STD_LOGIC_VECTOR(7 downto 0);
        data_out: OUT STD_LOGIC_VECTOR(5 downto 0)
    );
END sync_rom;

ARCHITECTURE rtl OF sync_rom IS
BEGIN
PROCESS (clock)
    BEGIN
    IF rising_edge (clock) THEN
        CASE address IS
```

```
            WHEN "00000000" => data_out <= "101111";
            WHEN "00000001" => data_out <= "110110";
            ...
            WHEN "11111110" => data_out <= "000001";
            WHEN "11111111" => data_out <= "101010";
            WHEN OTHERS => data_out <= "101111";
        END CASE;
    END IF;
    END PROCESS;
END rtl;
```

Shift Registers—Inferring the altshift_taps Megafunction from HDL Code

To infer shift registers, synthesis tools detect a group of shift registers of the same length and convert them to an `altshift_taps` megafunction. To be detected, all the shift registers must have the following characteristics:

- Use the same clock and clock enable
- Do not have any other secondary signals
- Have equally spaced taps that are at least three registers apart

When you are using a formal verification flow, Altera recommends that you create shift register blocks in separate entities or modules containing only the shift register logic, because you may need to treat the entity or module as a black box during formal verification.



Because formal verification tools do not support shift register megafunctions, the Quartus II integrated synthesis does not infer the `altshift_taps` megafunction when a formal verification tool is selected. You can select EDA tools for use with your Quartus II project on the **EDA Tool Settings** page of the **Settings** dialog box.

Synthesis software recognizes shift registers only for device families that have dedicated RAM blocks and the software uses certain guidelines to determine the best implementation. The following guidelines are followed in Quartus II integrated synthesis and also are generally followed by other EDA tools:

- For FLEX® 10K and ACEX® 1K devices, the software does not infer `altshift_taps` megafunctions because FLEX 10K and ACEX 1K devices have a relatively small amount of dedicated memory.
- For APEX™ 20K and APEX II devices, the software infers the `altshift_taps` megafunction only if the shift register has more than a total of 128 bits. Smaller shift registers typically do not benefit from implementation in dedicated memory.

- For Arria GX devices, and the Stratix and Cyclone series devices, the software determines whether to infer the `altshift_taps` megafunction based on the width of the registered bus (*W*), the length between each tap (*L*), and the number of taps (*N*).
  - If the registered bus width is one (*W = 1*), the software infers `altshift_taps` if the number of taps times the length between each tap is greater than or equal to 64 ($N \times L \geq 64$).
  - If the registered bus width is greater than one (*W > 1*), the software infers `altshift_taps` if the registered bus width times the number of taps times the length between each tap is greater than or equal to 32 ($W \times N \times L \geq 12$).

If the length between each tap (*L*) is not a power of two, the software uses more logic to decode the read and write counters. This situation occurs because for different sizes of shift registers, external decode logic that uses logic elements (LEs) or Adaptive Logic Modules (ALMs) is required to implement the function. This decode logic eliminates the performance and utilization advantages of implementing shift registers in memory.

The registers that the software maps to the `altshift_taps` megafunction and places in RAM are not available in a Verilog HDL or VHDL output file for simulation tools because their node names do not exist after synthesis.

Simple Shift Register

The code sample shown in Example 6–24 and Example 6–25 show a simple, single-bit wide, 64-bit long shift register. The synthesis software implements the register (*W* = 1 and *M* = 64) in an `altshift_taps` megafunction for supported devices. If the length of the register is less than 64 bits, the software implements the shift register in logic.

Example 6–24. Verilog HDL Single-Bit Wide, 64-Bit Long Shift Register

```
module shift_1x64 (clk, shift, sr_in, sr_out);
    input clk, shift;
    input sr_in;
    output sr_out;

    reg [63:0] sr;

    always @ (posedge clk)
    begin
        if (shift == 1'b1)
        begin
            sr[63:1] <= sr[62:0];
            sr[0] <= sr_in;
        end
    end
    assign sr_out = sr[63];
endmodule
```

Example 6–25. VHDL Single-Bit Wide, 64-Bit Long Shift Register

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shift_1x64 IS
   PORT (
      clk: IN STD_LOGIC;
      shift: IN STD_LOGIC;
      sr_in: IN STD_LOGIC;
      sr_out: OUT STD_LOGIC
   );
   END shift_1x64;

ARCHITECTURE arch OF shift_1x64 IS
   TYPE sr_length IS ARRAY (63 DOWNTO 0) OF STD_LOGIC;
   SIGNAL sr: sr_length;
BEGIN
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT and clk = '1') THEN
         IF (shift = '1') THEN
            sr(63 DOWNTO 1) <= sr(62 DOWNTO 0);
            sr(0) <= sr_in;
         END IF;
      END IF;
   END PROCESS;
   sr_out <= sr(63);
END arch;
```

Shift Register with Evenly Spaced Taps

The code samples shown in Examples 6–26 and 6–27 show a Verilog HDL and VHDL 8-bit wide, 64-bit long shift register ($W > 1$ and $M = 64$) with evenly spaced taps at 15, 31, and 47. The synthesis software implements this function in a single `altshift_taps` megafunction and maps it to RAM in supported devices.

Example 6–26. Verilog HDL 8-Bit Wide, 64-Bit Long Shift Register with Evenly Spaced Taps

```
module shift_8x64_taps (clk, shift, sr_in, sr_out, sr_tap_one, sr_tap_two, sr_tap_three );
    input clk, shift;
    input [7:0] sr_in;
    output [7:0] sr_tap_one, sr_tap_two, sr_tap_three, sr_out;

    reg [7:0] sr [63:0];
    integer n;

    always @ (posedge clk)
    begin
        if (shift == 1'b1)
        begin
            for (n = 63; n>0; n = n-1)
            begin
                sr[n] <= sr[n-1];
            end
```

```
            sr[0] <= sr_in;
        end

    end
    assign sr_tap_one = sr[15];
    assign sr_tap_two = sr[31];
    assign sr_tap_three = sr[47];
    assign sr_out = sr[63];
endmodule
```

Example 6–27. VHDL 8-Bit Wide, 64-Bit Long Shift Register with Evenly Spaced Taps

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shift_8x64_taps IS
    PORT (
        clk: IN STD_LOGIC;
        shift: IN STD_LOGIC;
        sr_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_one: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_two : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_three: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_out: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
    END shift_8x64_taps;

ARCHITECTURE arch OF shift_8x64_taps IS
    SUBTYPE sr_width IS STD_LOGIC_VECTOR(7 DOWNTO 0);
    TYPE sr_length IS ARRAY (63 DOWNTO 0) OF sr_width;
    SIGNAL sr: sr_length;
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'EVENT and clk = '1') THEN
            IF (shift = '1') THEN
                sr(63 DOWNTO 1) <= sr(62 DOWNTO 0);
                sr(0) <= sr_in;
            END IF;
        END IF;
    END PROCESS;
    sr_tap_one <= sr(15);
    sr_tap_two <= sr(31);
    sr_tap_three <= sr(47);
    sr_out <= sr(63);
END arch;
```

Coding Guidelines for Registers and Latches

This section provides device-specific coding recommendations for Altera registers and latches. Understanding the architecture of the target Altera device helps ensure that your code provides the expected results and achieves the optimal quality of results.

This section provides guidelines in the following areas:

- "Register Power-Up Values in Altera Devices"
- "Secondary Register Control Signals Such as Clear and Clock Enable" on page 6–39
- "Latches" on page 6–43

Register Power-Up Values in Altera Devices

Registers in the device core always power up to a low (`0`) logic level on all Altera devices. However, there are ways to implement logic such that registers behave as if they were powering up to a high (`1`) logic level.

If you use a preset signal on a device that does not support presets in the register architecture, then your synthesis tool may convert the preset signal to a clear signal, which requires synthesis to perform an optimization referred to as `NOT` gate push-back. `NOT` gate push-back adds an inverter to the input and the output of the register so that the reset and power-up conditions will appear to be high but the device operates as expected. In this case, your synthesis tool may issue a message informing you about the power-up condition. The register itself powers up low, but the register output is inverted so the signal that arrives at all destinations is high.

Due to these effects, if you specify a non-zero reset value, you may cause your synthesis tool to use the asynchronous clear (`aclr`) signals available on the registers to implement the high bits with `NOT` gate push-back. In that case, the registers look as though they power up to the specified reset value. You see this behavior, for example, if your design targets FLEX 10KE or ACEX devices.

When a load signal is available in the device, your synthesis tools can implement a reset of 1 or 0 value by using an asynchronous load of `1` or `0`. When the synthesis tool uses an asynchronous load signal, it is not performing `NOT` gate push-back, so the registers power up to a `0` logic level.



For additional details, refer to the appropriate device family handbook or the appropriate handbook of the Altera website at www.altera.com.

Designers typically use an explicit reset signal for the design, which forces all registers into their appropriate values after reset but not necessarily at power-up. You can create your design such that the asynchronous reset allows the board to operate in a safe condition and then you can bring up the design with the reset active. This is a good practice so you do not depend on the power-up conditions of the device.

You can make the your design more stable and avoid potential glitches by synchronizing external or combinational logic of the device architecture before you drive the asynchronous control ports of registers.



For additional information about good synchronous design practices, refer to the *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*.

If you want to force a particular power-up condition for your design, use the synthesis options available in your synthesis tool. With Quartus II integrated synthesis, you can apply the **Power-Up Level** logic option. You can also apply the option with an `altera_attribute` assignment in your source code. Using this option forces synthesis to perform `NOT` gate push-back because synthesis tools cannot actually change the power-up states of core registers.

You can apply the Quartus II integrated synthesis **Power-Up Level** assignment to a specific register or to a design entity, module or subdesign. If you do so, every register in that block receives the value. Registers power up to 0 by default; therefore you can use this assignment to force all registers to power up to 1 using `NOT` gate push-back.



Be aware that using `NOT` gate push-back as a global assignment could slightly degrade the quality of results due to the number of inverters that are needed. In some situations, issues are caused by enable or secondary control logic inference. It may also be more difficult to migrate such a design to an ASIC or a HardCopy® device. You can simulate the power-up behavior in a functional simulation if you use initialization.



The **Power-Up Level** option and the `altera_attribute` assignment are described in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Some synthesis tools can also read the default or initial values for registered signals and implement this behavior in the device. For example, Quartus II integrated synthesis converts default values for registered signals into Power-Up Level settings. That way, the synthesized behavior matches the power-up state of the HDL code during a functional simulation.

For example, the code samples in Example 6–28 and Example 6–29 both infer a register for q and set its power-up level to high (while the reset value is 0).

Example 6–28. Verilog Register with Reset and High Power-Up Value

```
reg q = 1'b1;

always @ (posedge clk or posedge aclr)
begin
    if (aclr)
        q <= 1'b0;
    else
        q <= d;
end
```

Example 6–29. VHDL Register with Reset and High Power-Up Level

```
SIGNAL q : STD_LOGIC := '1'; -- q has a default value of '1'

PROCESS (clk, reset)
BEGIN
   IF (reset = '1') THEN
      q <= '0';
   ELSIF (rising_edge(clk)) THEN
      q <= d;
   END IF;
END PROCESS;
```

Secondary Register Control Signals Such as Clear and Clock Enable

FPGA device architectures contain registers, also known as "flipflops". The registers in Altera FPGAs provide a number of secondary control signals (such as clear and enable signals) that you can use to implement control logic for each register without using extra logic cells. Device families vary in their support for secondary signals, so consult the device family data sheet to verify which signals are available in your target device.

To make the most efficient use of the signals in the device, your HDL code should match the device architecture as closely as possible. The control signals have a certain priority due to the nature of the architecture, so your HDL code should follow that priority where possible.

Your synthesis tool can emulate any control signals using regular logic, so getting functionally correct results is always possible. However, if your design requirements are flexible in terms of which control signals are used and in what priority, match your design to the target device architecture to achieve the most efficient results. If the priority of the signals in your design is not the same as that of the target architecture, then extra logic may be required to implement the control signals. This extra logic uses additional device resources, and can cause additional delays for the control signals.

In addition, there are certain cases where using logic other than the dedicated control logic in the device architecture can have a larger impact. For example, the clock enable signal has priority over the synchronous reset or clear signal in the device architecture. The clock enable turns off the clock line in the logic array block (LAB), and the clear signal is synchronous. So in the device architecture, the synchronous clear takes effect only when a clock edge occurs.

If you code a register with a synchronous clear signal that has priority over the clock enable signal, the software must emulate the clock enable functionality using data inputs to the registers. Because the signal does not use the clock enable port of a register, you cannot apply a Clock Enable Multicycle constraint. In this case, following the priority of signals available in the device is clearly the best choice for the priority of these control signals, and using a different priority causes unexpected results with an assignment to the clock enable signal.



The priority order for secondary control signals in Altera devices differs from the order for other vendors' devices. If your design requirements are flexible regarding priority, verify that the secondary control signals meet design performance requirements when migrating designs between FPGA vendors and try to match your target device architecture to achieve the best results.

The signal order is the same for all Altera device families, although as noted previously, not all device families provide every signal. The following priority order is observed:

1. Asynchronous Clear, `aclr`—highest priority
2. Preset, `pre`
3. Asynchronous Load, `aload`
4. Enable, `ena`
5. Synchronous Clear, `sclr`
6. Synchronous Load, `sload`
7. Data In, `data`—lowest priority

The following examples provide Verilog HDL and VHDL code that creates a register with the `aclr`, `aload`, and `ena` control signals.



The Verilog HDL example (Example 6–30) does not have `adata` on the sensitivity list, but the VHDL example (Example 6–31) does. This is a limitation of the Verilog HDL language—there is no way to describe an asynchronous load signal (in which `q` toggles if `adata` toggles while `aload` is high). All synthesis tools should infer an `aload` signal from this construct despite this limitation. When they perform such inference, you may see information or warning messages from the synthesis tool.

Example 6–30. Verilog HDL D-Type Flipflop (Register) with ena, aclr and aload Control Signals

```
module dff_control(clk, aclr, aload, ena, data, adata, q);
    input clk, aclr, aload, ena, data, adata;
    output q;

    reg q;

    always @ (posedge clk or posedge aclr or posedge aload)
    begin
        if (aclr)
            q <= 1'b0;
        else if (aload)
            q <= adata;
        else if (ena)
            q <= data;
    end
endmodule
```

Example 6–31. VHDL D-Type Flipflop (Register) with ena, aclr and aload Control Signals

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff_control IS
    PORT (
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      aload: IN STD_LOGIC;
      adata: IN STD_LOGIC;
      ena: IN STD_LOGIC;
      data: IN STD_LOGIC;
      q: OUT STD_LOGIC
    );
END dff_control;

ARCHITECTURE rtl OF dff_control IS
BEGIN
    PROCESS (clk, aclr, aload, adata)
    BEGIN
        IF (aclr = '1') THEN
          q <= '0';
        ELSIF (aload = '1') THEN
          q <= adata;
        ELSE
            IF (clk = '1' AND clk'event) THEN
                IF (ena ='1') THEN
                q <= data;
                END IF;
            END IF;
        END IF;
    END PROCESS;
END rtl;
```

The preset signal is not available in many device families, so the preset signal is not included in the examples.

Creating many registers with different `sload` and `sclr` signals can make packing the registers into LABs difficult for the Quartus II Fitter because the `sclr` and `sload` signals are LAB-wide signals. In addition, using the LAB-wide `sload` signal prevents the Fitter from packing registers using the quick feedback path in the device architecture, which means that some registers cannot be packed with other logic.

Synthesis tools typically restrict use of `sload` and `sclr` signals to cases in which there are enough registers with common signals to allow good LAB packing. Using the LUT to implement the signals is always more flexible if it is available. Because different device families offer different numbers of control signals, inference of these signals is also device-specific. For example, Stratix II devices have more flexibility than Stratix devices with respect to secondary control signals, so synthesis tools might infer more `sload` and `sclr` signals for Stratix II devices.

If you use these additional control signals, use them in the priority order that matches the device architecture. To achieve the most efficient results, ensure the sclr signal has a higher priority than the sload signal in the same way that aclr has higher priority than aload in the previous examples. Remember that the register signals are not inferred unless the design meets the conditions described previously. However, if your HDL described the desired behavior, the software always implements logic with the correct functionality.

In Verilog HDL, the following code for sload and sclr could replace the if (ena) q <= data; statements in the Verilog HDL example shown in Example 6–30 on page 6–41 (after adding the control signals to the module declaration).

Example 6–32. Verilog HDL sload and sclr Control Signals

```
if  (ena) begin
   if  (sclr)
      q <= 1'b0;
   else if  (sload)
      q <= sdata;
   else
      q <= data;
end
```

In VHDL, the following code for sload and sclr could replace the IF (ena ='1') THEN q <= data; END IF; statements in the VHDL example shown in Example 6–31 on page 6–42 (after adding the control signals to the entity declaration).

Example 6–33. VHDL sload and sclr Control Signals

```
IF (ena ='1') THEN
   IF (sclr = '1') THEN
      q <= '0';
   ELSIF (sload = '1') THEN
      q <= sdata;
   ELSE
      q <= data;
    END IF;
END IF;
```

Latches

A latch is a small combinational loop that holds the value of a signal until a new value is assigned.



Altera recommends that you design without the use of latches whenever possible.

For additional information about the issues involved in designing with latches and all combinational loops, refer to the *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*.

Latches can be inferred from HDL code when you did not intend to use a latch as detailed in "Unintentional Latch Generation". If you do intend to infer a latch, it is important to infer it correctly to guarantee correct device operation as detailed in "Inferring Latches Correctly" on page 6–45.

Unintentional Latch Generation

When you are designing combinational logic, certain coding styles can create an unintentional latch. For example, when `CASE` or `IF` statements do not cover all possible input conditions, latches may be required to hold the output if a new output value is not assigned. Check your synthesis tool messages for references to inferred latches. If your code unintentionally creates a latch, make code changes to remove the latch.

Latches have limited support in formal verification tools. Therefore, ensure that you do not infer latches unintentionally. For example, an incomplete `CASE` statement may create a latch when you are using formal verification in your design flow.

The `full_case` attribute can be used in Verilog HDL designs to treat unspecified cases as don't care values (`X`). However, using the `full_case` attribute can cause simulation mismatches because this attribute is a synthesis-only attribute, so simulation tools still treat the unspecified cases as latches.

Refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook* for more information about using attributes in your synthesis tool. The *Quartus II Integrated Synthesis* chapter provides an example explaining possible simulation mismatches.

Omitting the final `ELSE` or `WHEN OTHERS` clause in an `IF` or `CASE` statement can also generate a latch. Don't care (`X`) assignments on the default conditions are useful in preventing latch generation. For the best logic optimization, assign the default `CASE` or final `ELSE` value to don't care (`X`) instead of a logic value.

The VHDL sample code shown in Example 6–34 prevents unintentional latches. Without the final `ELSE` clause, this code creates unintentional latches to cover the remaining combinations of the `sel` inputs. When you are targeting a Stratix device with this code, omitting the final `ELSE` condition can cause the synthesis software to use up to six LEs, instead of the three it uses with the `ELSE` statement. Additionally, assigning the final `ELSE` clause to `1` instead of `X` can result in slightly more LEs because the synthesis software cannot perform as much optimization when you specify a constant value compared to a don't care value.

Example 6–34. VHDL Code Preventing Unintentional Latch Creation

```
LIBRARY ieee;
USE IEEE.std_logic_1164.all;

ENTITY nolatch IS
    PORT (a,b,c: IN STD_LOGIC;
        sel: IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        oput: OUT STD_LOGIC);
END nolatch;

ARCHITECTURE rtl OF nolatch IS
BEGIN
    PROCESS (a,b,c,sel) BEGIN
        IF sel = "00000" THEN
            oput <= a;
        ELSIF sel = "00001" THEN
            oput <= b;
        ELSIF sel = "00010" THEN
            oput <= c;
        ELSE               --- Prevents latch inference
            oput <= ''X'; --/
        END IF;
    END PROCESS;
END rtl;
```

Inferring Latches Correctly

Synthesis tools can infer a latch that does not exhibit the glitch and timing hazard problems typically associated with combinational loops.



Any use of latches generates warnings and is flagged if the design is migrated to a HardCopy structured ASIC. In addition, timing analysis does not completely model latch timing in some cases. Do not use latches unless you are very certain that your design requires it, and you fully understand the impact of using the latches.

When using Quartus II integrated synthesis, latches that are inferred by the software are reported in the **User-Specified and Inferred Latches** section of the Compilation Report. This report indicates whether the latch is considered safe and free of timing hazards.

If a latch or combinational loop in your design is not listed in the **User-Specified and Inferred Latches** report, it means that it was not inferred as a safe latch by the software and is not considered glitch-free.

All combinational loops listed in the **Analysis & Synthesis Logic Cells Representing Combinational Loops** table in the **Compilation Report** are at risk of timing hazards. These entries indicate possible problems with your design that you should investigate. However, it is possible to have

a correct design that includes combinational loops. For example, it is possible that the combinational loop cannot be sensitized. This can occur in cases where there is an electrical path in the hardware, but either the designer knows that the circuit will never encounter data that causes that path to be activated, or the surrounding logic is set up in a mutually exclusive manner that prevents that path from ever being sensitized, independent of the data input.

For macrocell-based devices such as MAX® 7000AE and MAX 3000A, all data (D-type) latches and set-reset (S-R) latches listed in the **Analysis & Synthesis User-Specified and Inferred Latches** table have an implementation free of timing hazards such as glitches. The implementation includes a cover term to ensure there is no glitching, and includes a single macrocell in the feedback loop.

For 4-input LUT-based devices such as Stratix devices, the Cyclone series, and MAX II devices, all latches in the **User-Specified and Inferred Latches** table with a single LUT in the feedback loop are free of timing hazards when a single input changes. Because of the hardware behavior of the LUT, the output does not glitch when a single input toggles between two values that are supposed to produce the same output value. For example, a D-type input toggling when the enable input is inactive, or a set input toggling when a reset input with higher priority is active. This hardware behavior of the LUT means that no cover term is needed for a loop around a single LUT. The Quartus II software uses a single LUT in the feedback loop whenever possible. A latch that has data, enable, set, and reset inputs in addition to the output fed back to the input cannot be implemented in a single 4-input LUT. If the Quartus II software cannot implement the latch with a single-LUT loop because there are too many inputs, then the **User-Specified and Inferred Latches** table indicates that the latch is not free of timing hazards.

For 6-input LUT-based devices, the software can implement all latch inputs with a single adaptive look-up table (ALUT) in the combinational loop. Therefore, all latches in the **User-Specified and Inferred Latches** table are free of timing hazards when a single input changes.

If a latch is listed as a safe latch, other Quartus II optimizations, such as physical synthesis netlist optimizations in the Fitter, maintain the hazard-free performance.

To ensure hazard-free behavior, only one control input may change at a time. Changing two inputs simultaneously, such as deasserting set and reset at the same time, or changing data and enable at the same time, can produce incorrect behavior in any latch.

Quartus II integrated synthesis infers latches from `always` blocks in Verilog HDL and `process` statements in VHDL, but not from continuous assignments in Verilog HDL or concurrent signal assignments in VHDL. These rules are the same as for register inference. The software infers registers or flipflops only from `always` blocks and `process` statements.

The Verilog HDL code sample shown in Example 6–35 infers a S-R latch correctly in the Quartus II software.

Example 6–35. Verilog HDL Set-Reset Latch

```
module simple_latch (
   input SetTerm,
   input ResetTerm,
   output reg LatchOut
   );

   always @ (SetTerm or ResetTerm) begin
      if (SetTerm)
         LatchOut = 1'b1
      else if (ResetTerm)
         LatchOut = 1'b0
   end
endmodule
```

The VHDL code sample shown in Example 6–36 infers a D-type latch correctly in the Quartus II software.

Example 6–36. VHDL Data Type Latch

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY simple_latch IS
   PORT (
      enable, data    : IN STD_LOGIC;
      q               : OUT STD_LOGIC
   );
END simple_latch;

ARCHITECTURE rtl OF simple_latch IS
BEGIN

latch : PROCESS (enable, data)
   BEGIN
      IF (enable = '1') THEN
         q <= data;
      END IF;
   END PROCESS latch;
END rtl;
```

The following example shows a Verilog HDL continuous assignment that does not infer a latch in the Quartus II software. The behavior is similar to a latch, but it may not function correctly as a latch and its timing is not analyzed as a latch.

```
assign latch_out = (~en & latch_out) | (en & data);
```

Quartus II integrated synthesis also creates safe latches when possible for instantiations of the `lpm_latch` megafunction. You can use this megafunction to create a latch with any combination of data, enable, set, and reset inputs. The same limitations apply for creating safe latches as for inferring latches from HDL code.

Inferring the Altera `lpm_latch` function in another synthesis tool ensures that the implementation is also recognized as a latch in the Quartus II software. If a third-party synthesis tool implements a latch using the `lpm_latch` megafunction, then the Quartus II integrated synthesis lists the latch in the **User-Specified and Inferred Latches** table in the same way as it lists latches created in HDL source code. The coding style necessary to produce an `lpm_latch` implementation may depend on your synthesis tool. Some third-party synthesis tools list the number of `lpm_latch` functions that are inferred.

For LUT-based families, the Fitter uses global routing for control signals including signals that Analysis and Synthesis identifies as latch enables. In some cases the global insertion delay may decrease the timing performance. If necessary, you can turn off the Quartus II **Global Signal** logic option to manually prevent the use of global signals. Global latch enables are listed in the **Global & Other Fast Signals** table in the Compilation Report.

General Coding Guidelines

This section helps you understand how synthesis tools map various types of HDL code into the target Altera device. Following Altera recommended coding styles, and in some cases designing logic structures to match the appropriate device architecture, can provide significant improvements in the design's quality of results.

This section provides coding guidelines for the following logic structures:

- "Tri-State Signals". This section explains how to create tri-state signals for bidirectional I/O pins.
- "Adder Trees" on page 6–50. This section explains the different coding styles that lead to optimal results for devices with 4-input look-up tables and 6-input adaptive look-up tables.
- "State Machines" on page 6–52. This section helps ensure the best results when you use state machines.

- "Multiplexers" on page 6–60. This section explains how multiplexers can be synthesized for 4-input LUT devices, addresses common problems, and provides guidelines to achieve optimal resource utilization.
- "Cyclic Redundancy Check Functions" on page 6–69. This section provides guidelines for getting good results when designing CRC functions.
- "Comparators" on page 6–71. This section explains different comparator implementations and provides suggestions for controlling the implementation.
- "Counters" on page 6–73. This section provides guidelines to ensure your counter design targets the device architecture optimally.

Tri-State Signals

When you are targeting Altera devices, you should use tri-state signals only when they are attached to top-level bidirectional or output pins. Avoid lower level bidirectional pins, and avoid using the Z logic value unless it is driving an output or bidirectional pin.

Synthesis tools implement designs with internal tri-state signals correctly in Altera devices using multiplexer logic, but Altera does not recommend this coding practice.



In hierarchical block-based or incremental design flows, a hierarchical boundary cannot contain any bidirectional ports, unless the lower level bidirectional port is connected directly through the hierarchy to a top-level output pin without connecting to any other design logic. If you use boundary tri-states in a lower level block, synthesis software must push the tri-states through the hierarchy to the top-level to make use of the tri-state drivers on output pins of Altera devices. Because pushing tri-states requires optimizing through hierarchies, lower level tri-states are restricted with block-based design methodologies.

The code examples shown in Examples 6–37 and 6–38 show Verilog HDL and VHDL code that creates tri-state bidirectional signals.

Example 6–37. Verilog HDL Tri-State Signal

```
module tristate (myinput, myenable, mybidir);
    input myinput, myenable;
    inout mybidir;
    assign mybidir = (myenable ? myinput : 1'bZ);
endmodule
```

Example 6–38. VHDL Tri-State Signal

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY tristate IS
PORT (
   mybidir : INOUT STD_LOGIC;
   myinput : IN STD_LOGIC;
   myenable : IN STD_LOGIC
    );
END tristate;

ARCHITECTURE rtl OF tristate IS
BEGIN
    mybidir <= 'Z' WHEN (myenable = '0') ELSE myinput;
END rtl;
```

Adder Trees

Structuring adder trees appropriately to match your targeted Altera device architecture can result in significant performance and density improvements. A good example of an application using a large adder tree is a finite impulse response (FIR) correlator. Using a pipelined binary or ternary adder tree appropriately can greatly improve the quality of your results.

This section explains why coding recommendations are different for Altera 4-input LUT devices and 6-input LUT devices.

Architectures with 4-Input LUTs in Logic Elements

Architectures such as Stratix devices and the Cyclone series, APEX series, and FLEX series devices contain 4-input LUTs as the standard combinational structure in the LE.

If your design can tolerate pipelining, the fastest way to add three numbers `A`, `B`, and `C` in devices that use 4-input lookup tables is to add `A + B`, register the output, and then add the registered output to `C`. Adding `A + B` takes one level of logic (one bit is added in one LE), so this runs at full clock speed. This can be extended to as many numbers as desired.

In the code sample shown in Example 6–39, five numbers A, B, C, D, and E are added. Adding five numbers in devices that use 4-input lookup tables requires four adders and three levels of registers for a total of 64 LEs (for 16-bit numbers).

Example 6–39. Verilog HDL Pipelined Binary Tree

```
module binary_adder_tree (A, B, C, D, E, CLK, OUT);
    parameter WIDTH = 16;
    input [WIDTH-1:0] A, B, C, D, E;
    input CLK;
    output [WIDTH-1:0] OUT;

    wire [WIDTH-1:0] sum1, sum2, sum3, sum4;
    reg [WIDTH-1:0] sumreg1, sumreg2, sumreg3, sumreg4;
    // Registers

    always @ (posedge CLK)
        begin
            sumreg1 <= sum1;
            sumreg2 <= sum2;
            sumreg3 <= sum3;
            sumreg4 <= sum4;
        end

    // 2-bit additions
    assign sum1 = A + B;
    assign sum2 = C + D;
    assign sum3 = sumreg1 + sumreg2;
    assign sum4 = sumreg3 + E;
    assign OUT = sumreg4;
endmodule
```

Architectures with 6-Input LUTs in Adaptive Logic Modules

Newer high-performance Altera device families use a 6-input LUT in their basic logic structure, so these devices benefit from a different coding style from the previous example presented for 4-input LUTs. Specifically, in these devices, ALMs can simultaneously add three bits. Therefore, the tree in the previous example must be two levels deep and contain just two add-by-three inputs instead of four add-by-two inputs.

Although the code in the previous example compiles successfully for 6-input LUT devices, the code is inefficient and does not take advantage of the 6-input adaptive look-up table (ALUT). By restructuring the tree as a ternary tree, the design becomes much more efficient, significantly improving density utilization. Therefore, when you are targeting with ALUTs and ALMs, large pipelined binary adder trees designed for 4-input LUT architectures should be rewritten to take advantage of the advanced device architecture.

Example 6–40 uses just 32 ALUTs in a Stratix II device—more than a 4:1 advantage over the number of LUTs in the prior example implemented in a Stratix device.



You cannot pack a LAB full when using this type of coding style because of the number of LAB inputs. However, in a typical design, the Quartus II Fitter can pack other logic into each LAB to take advantage of the unused ALMs.

Example 6–40. Verilog HDL Pipelined Ternary Tree

```
module ternary_adder_tree (A, B, C, D, E, CLK, OUT);
    parameter WIDTH = 16;
    input [WIDTH-1:0] A, B, C, D, E;
    input CLK;
    output [WIDTH-1:0] OUT;

    wire [WIDTH-1:0] sum1, sum2;
    reg [WIDTH-1:0] sumreg1, sumreg2;
    // Registers

    always @ (posedge CLK)
        begin
            sumreg1 <= sum1;
            sumreg2 <= sum2;
        end

    // 3-bit additions
    assign sum1 = A + B + C;
    assign sum2 = sumreg1 + D + E;
    assign OUT = sumreg2;
endmodule
```

These examples show pipelined adders, but partitioning your addition operations can help you achieve better results in nonpipelined adders as well. If your design is not pipelined, a ternary tree provides much better performance than a binary tree. For example, depending on your synthesis tool, the HDL code `sum = (A + B + C) + (D + E)` is more likely to create the optimal implementation of a 3-input adder for `A + B + C` followed by a 3-input adder for `sum1 + D + E` than the code without the parentheses. If you do not add the parentheses, the synthesis tool may partition the addition in a way that is not optimal for the architecture.

State Machines

Synthesis tools can recognize and encode Verilog HDL and VHDL state machines during synthesis. This section presents guidelines to ensure the best results when you use state machines. Ensuring that your synthesis tool recognizes a piece of code as a state machine allows the tool to recode the state variables to improve the quality of results, and allows the tool to

use the known properties of state machines to optimize other parts of the design. When synthesis recognizes a state machine it is often able to improve the design area and performance.

To achieve the best results on average, synthesis tools often use one-hot encoding for FPGA devices and minimal-bit encoding for CPLD devices, although the choice of implementation can vary for different state machines and different devices. Refer to your synthesis tool documentation for specific ways to control the manner in which state machines are encoded.



For information about state machine encoding in Quartus II integrated synthesis, refer to the *State Machine Processing* section in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

To ensure proper recognition and inference of state machines and to improve the quality of results, Altera recommends that you observe the following guidelines, which apply to both Verilog HDL and VHDL:

- Assign default values to outputs derived from the state machine so that synthesis does not generate unwanted latches.
- Separate the state machine logic from all arithmetic functions and data paths, including assigning output values.
- If your design contains an operation that is used by more than one state, define the operation outside the state machine and cause the output logic of the state machine to use this value.
- Use a simple asynchronous or synchronous reset to ensure a defined power-up state. If your state machine design contains more elaborate reset logic, such as both an asynchronous reset and an asynchronous load, the Quartus II software generates regular logic rather than inferring a state machine.

If a state machine enters an illegal state due to a problem with the device, the design likely ceases to function correctly until the next reset of the state machine. Synthesis tools do not provide for this situation by default. The same issue applies to any other registers if there is some kind of fault in the system. A `default` or `when others` clause does not affect this operation, assuming that your design never deliberately enters this state. Synthesis tools remove any logic generated by a default state if it is not reachable by normal state machine operation.

Many synthesis tools (including Quartus II integrated synthesis) have an option to implement a safe state machine. The software inserts extra logic to detect an illegal state and force the state machine's transition to the reset state. It is commonly used when the state machine can enter an

illegal state. The most common cause of this situation is a state machine that has control inputs that come from another clock domain, such as the control logic for a dual-clock FIFO.

Of course this option protects only state machines, and all other registers in the design are not protected this way.



For additional information about tool-specific options for implementing state machines, refer to the tool vendor's documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

The following two sections, "Verilog HDL State Machines" and "VHDL State Machines" on page 6–58, describe additional language-specific guidelines and coding examples.

Verilog HDL State Machines

To ensure proper recognition and inference of Verilog HDL state machines, observe the following additional Verilog HDL guidelines. Some of these guidelines may be specific to Quartus II integrated synthesis. Refer to your synthesis tool documentation for specific coding recommendations.

If the state machine is not recognized by the synthesis software (such as Quartus II integrated synthesis), the state machine is implemented as regular logic gates and registers and the state machine is not listed as a state machine in the **Analysis & Synthesis** section of the Quartus II Compilation Report. In this case, the software does not perform any of the optimizations that are specific to state machines.

- If you are using the SystemVerilog standard, use enumerated types to describe state machines (as shown in the "SystemVerilog State Machine Coding Example" on page 6–57).
- Represent the states in a state machine with the `parameter` data types in Verilog-1995 and -2001 and use the parameters to make state assignments (as shown below in the "Verilog HDL State Machine Coding Example"). This implementation makes the state machine easier to read and reduces the risk of errors during coding.



Altera recommends against the direct use of integer values for state variables such as `next_state <= 0`. However, using an integer does not prevent inference in the Quartus II software.

- No state machine is inferred in the Quartus II software if the state transition logic uses arithmetic similar to that shown in the following example:

```
case (state)
    0: begin
        if (ena) next_state <= state + 2;
        else next_state <= state + 1;
    end
    1: begin
    ...
endcase
```

- No state machine is inferred in the Quartus II software if the state variable is an output.
- No state machine is inferred in the Quartus II software for signed variables

Verilog HDL State Machine Coding Example

The following module `verilog_fsm` is an example of a typical Verilog HDL state machine implementation (Example 6–41).

This state machine has five states. The asynchronous reset sets the variable state to `state_0`. The sum of `in_1` and `in_2` is an output of the state machine in `state_1` and `state_2`. The difference (`in_1` – `in_2`) is also used in `state_1` and `state_2`. The temporary variables `tmp_out_0` and `tmp_out_1` store the sum and the difference of `in_1` and `in_2`. Using these temporary variables in the various states of the state machine ensures proper resource sharing between the mutually exclusive states.

Example 6–41. Verilog-2001 State Machine

```
module verilog_fsm (clk, reset, in_1, in_2, out);
    input clk;
    input reset;
    input [3:0] in_1;
    input [3:0] in_2;output [4:0] out;
    parameter state_0 = 3'b000;
    parameter state_1 = 3'b001;
    parameter state_2 = 3'b010;
    parameter state_3 = 3'b011;
    parameter state_4 = 3'b100;

    reg [4:0] tmp_out_0, tmp_out_1, tmp_out_2;
    reg [2:0] state, next_state;

    always @ (posedge clk or posedge reset)
    begin
        if (reset)
            state <= state_0;
        else
```

```
            state <= next_state;
    end
    always @ (state or in_1 or in_2)
    begin
        tmp_out_0 = in_1 + in_2;
        tmp_out_1 = in_1 - in_2;
        case (state)
            state_0: begin
               tmp_out_2 <= in_1 + 5'b00001;
               next_state <= state_1;
            end
            state_1: begin
                if (in_1 < in_2) begin
                    next_state <= state_2;
                    tmp_out_2 <= tmp_out_0;
                end
                else begin
                    next_state <= state_3;
                    tmp_out_2 <= tmp_out_1;
                end
            end
            state_2: begin
                tmp_out_2 <= tmp_out_0 - 5'b00001;
                next_state <= state_3;
            end
            state_3: begin
                tmp_out_2 <= tmp_out_1 + 5'b00001;
                next_state <= state_0;
            end
            state_4:begin
                tmp_out_2 <= in_2 + 5'b00001;
                next_state <= state_0;
            end
            default:begin
                tmp_out_2 <= 5'b00000;
                next_state <= state_0;
            end
        endcase
    end
    assign out = tmp_out_2;
endmodule
```

An equivalent implementation of this state machine can be achieved by using `‘define` instead of the `parameter` data type, as follows:

```
‘define state_0 3'b000
‘define state_1 3'b001
‘define state_2 3'b010
‘define state_3 3'b011
‘define state_4 3'b100
```

In this case, the `state` and `next_state` assignments are assigned a `‘state_x` instead of a `state_x`, as shown in the following example:

```
next_state <= ‘state_3;
```



Although the `'define` construct is supported, Altera strongly recommends the use of the `parameter` data type because doing so preserves the state names throughout synthesis.

SystemVerilog State Machine Coding Example

The module `enum_fsm` shown in Example 6–42 is an example of a SystemVerilog state machine implementation that uses enumerated types. Altera recommends using this coding style to describe state machines in SystemVerilog.



In Quartus II integrated synthesis, the enumerated type that defines the states for the state machine must be of an unsigned integer type as shown in Example 6–42. If you do not specify the enumerated type as `int unsigned`, a signed `int` type is used by default. In this case, the Quartus II integrated synthesis synthesizes the design, but does not infer or optimize the logic as a state machine.

Example 6–42. SystemVerilog State Machine Using Enumerated Types

```
module enum_fsm (input clk, reset, input int data[3:0], output int o);

   enum int unsigned { S0 = 0, S1 = 2, S2 = 4, S3 = 8 } state, next_state;

   always_comb begin : next_state_logic
      next_state = S0;
      case(state)
        S0: next_state = S1;
        S1: next_state = S2;
        S2: next_state = S3;
        S3: next_state = S3;
      endcase
   end

   always_comb begin
      case(state)
         S0: o = data[3];
         S1: o = data[2];
         S2: o = data[1];
         S3: o = data[0];
      endcase
   end

   always_ff@(posedge clk or negedge reset) begin
      if(~reset)
         state <= S0;
      else
         state <= next_state;
   end
endmodule
```

VHDL State Machines

To ensure proper recognition and inference of VHDL state machines, represent the states in a state machine with enumerated types and use the corresponding types to make state assignments. This implementation makes the state machine easier to read and reduces the risk of errors during coding. If the state is not represented by an enumerated type, synthesis software (such as Quartus II integrated synthesis) does not recognize the state machine. Instead, the state machine is implemented as regular logic gates and registers and the state machine is not listed as a state machine in the **Analysis & Synthesis** section of the Quartus II Compilation Report. In this case, the software does not perform any of the optimizations that are specific to state machines.

VHDL State Machine Coding Example

The following entity, `vhdl_fsm`, is an example of a typical VHDL state machine implementation (Example 6–43).

This state machine has five states. The asynchronous reset sets the variable state to `state_0`. The sum of `in1` and `in2` is an output of the state machine in `state_1` and `state_2`. The difference (`in1 - in2`) is also used in `state_1` and `state_2`. The temporary variables `tmp_out_0` and `tmp_out_1` store the sum and the difference of `in1` and `in2`. Using these temporary variables in the various states of the state machine ensures proper resource sharing between the mutually exclusive states.

Example 6–43. VHDL State Machine

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY vhdl_fsm IS
   PORT(
      clk: IN STD_LOGIC;
      reset: IN STD_LOGIC;
      in1: IN UNSIGNED(4 downto 0);
      in2: IN UNSIGNED(4 downto 0);
      out_1: OUT UNSIGNED(4 downto 0)
      );
END vhdl_fsm;

ARCHITECTURE rtl OF vhdl_fsm IS
   TYPE Tstate IS (state_0, state_1, state_2, state_3, state_4);
   SIGNAL state: Tstate;
   SIGNAL next_state: Tstate;
BEGIN
   PROCESS(clk, reset)
   BEGIN
         IF reset = '1' THEN
            state <=state_0;
```

```
      ELSIF rising_edge(clk) THEN
            state <= next_state;
      END IF;
   END PROCESS;
   PROCESS (state, in1, in2)
      VARIABLE tmp_out_0: UNSIGNED (4 downto 0);
      VARIABLE tmp_out_1: UNSIGNED (4 downto 0);
   BEGIN
      tmp_out_0 := in1 + in2;
      tmp_out_1 := in1 - in2;
      CASE state IS
         WHEN state_0 =>
            out_1 <= in1;
            next_state <= state_1;
         WHEN state_1 =>
            IF (in1 < in2) then
               next_state <= state_2;
               out_1 <= tmp_out_0;
            ELSE
               next_state <= state_3;
               out_1 <= tmp_out_1;
            END IF;
         WHEN state_2 =>
            IF (in1 < "0100") then
               out_1 <= tmp_out_0;
            ELSE
               out_1 <= tmp_out_1;
            END IF;
               next_state <= state_3;
         WHEN state_3 =>
               out_1 <= "11111";
               next_state <= state_4;
         WHEN state_4 =>
               out_1 <= in2;
               next_state <= state_0;
         WHEN OTHERS =>
               out_1 <= "00000";
               next_state <= state_0;
      END CASE;
   END PROCESS;
END rtl;
```

Multiplexers

Multiplexers form a large portion of the logic utilization in many FPGA designs. By optimizing your multiplexer logic, you ensure the most efficient implementation in your Altera device. This section addresses common problems and provides design guidelines to achieve optimal resource utilization for multiplexer designs. The section also describes various types of multiplexers, and how they are implemented in the 4-input LUT found in many FPGA architectures, such as Altera's Stratix devices.



Stratix II and newer high-performance devices have 6-input ALUTs and are not specifically addressed here. Although many of the principles and techniques for optimization are similar, device utilization differs in the 6-input LUT devices. For example, these devices can implement wider multiplexers in one ALM than can be implemented in the 4-input LUT of an LE.

Quartus II Software Option for Multiplexer Restructuring

Quartus II integrated synthesis provides the **Restructure Multiplexers** logic option that extracts and optimizes buses of multiplexers during synthesis. In certain situations, this option automatically performs some of the recoding functions described in this section without changing the HDL code in your design. This option is on by default, when the Optimization technique is set to **Balanced** (the default for most device families) or set to **Area**.



For details, refer to the *Restructure Multiplexers* subsection in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Even with this Quartus II-specific option turned on, it is beneficial to understand how your coding style can be interpreted by your synthesis tool, and avoid the situations that can cause problems in your design.

Multiplexer Types

This first subsection addresses how multiplexers are created from various types of HDL code. `CASE` statements, `IF` statements, and state machines are all common sources of multiplexer logic in designs. These HDL structures create different types of multiplexers including binary multiplexers, selector multiplexers, and priority multiplexers. Understanding how multiplexers are created from HDL code and how they might be implemented during synthesis is the first step towards optimizing multiplexer structures for best results.

Binary Multiplexers

Binary multiplexers select inputs based on binary-encoded selection bits. Example 6–44 shows Verilog HDL code for two ways to describe a simple 4:1 binary multiplexer.

Example 6–44. Verilog HDL Binary-Encoded Multiplexers

```
case (sel)
   2'b00: z = a;
   2'b01: z = b;
   2'b10: z = c;
   2'b11: z = d;
endcase
```

A 4:1 binary multiplexer is efficiently implemented by using two 4-input LUTs. Larger binary multiplexers can be constructed that use the 4:1 multiplexer; constructing an *N*-input multiplexer (*N*:1 multiplexer) from a tree of 4:1 multiplexers can result in a structure using as few as 0.66*(*N* - 1) LUTs.

Selector Multiplexers

Selector multiplexers have a separate select line for each data input. The select lines for the multiplexer are one-hot encoded. Example 6–45 shows a simple Verilog HDL code example describing a one-hot selector multiplexer.

Example 6–45. Verilog HDL One-Hot-Encoded Case Statement

```
case (sel)
   4'b0001: z = a;
   4'b0010: z = b;
   4'b0100: z = c;
   4'b1000: z = d;
   default: z = 1'bx;
endcase
```

Selector multiplexers are commonly built as a tree of AND and OR gates. Using this scheme, two inputs can be selected using two select lines in a single 4-input LUT that uses two AND gates and an OR gate. The outputs of these LUTs can be combined with a wide OR gate. An *N*-input selector multiplexer of this structure requires at least 0.66*(*N*-0.5) LUTs, which is just slightly worse than the best binary multiplexer.

Priority Multiplexers

In priority multiplexers, the select logic implies a priority. The options to select the correct item must be checked in a specific order based on signal priority. These structures commonly are created from IF, ELSE, WHEN,

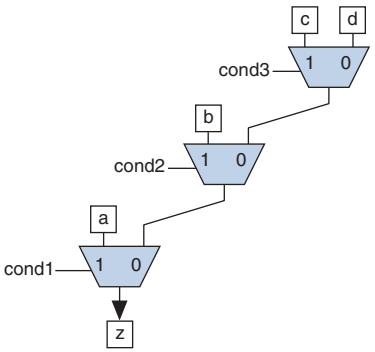
SELECT, and ? : statements in VHDL or Verilog HDL. The example VHDL code in Example 6–46 will probably result in the schematic implementation illustrated in Figure 6–2.

Example 6–46. VHDL IF Statement Implying Priority

```
IF cond1 THEN z <= a;
ELSIF cond2 THEN z <= b;
ELSIF cond3 THEN z <= c;
ELSE z <= d;
END IF;
```

The multiplexers shown in Figure 6–2 form a chain, evaluating each condition or select bit, one at a time.

Figure 6–2. Priority Multiplexer Implementation of an IF Statement



An *N*-input priority multiplexer uses a LUT for every 2:1 multiplexer in the chain, requiring *N*-1 LUTs. This chain of multiplexers generally increases delay because the critical path through the logic traverses every multiplexer in the chain.

To improve the timing delay through the multiplexer, avoid priority multiplexers if priority is not required. If the order of the choices is not important to the design, use a CASE statement to implement a binary or selector multiplexer instead of a priority multiplexer. If delay through the structure is important in a multiplexed design requiring priority, consider recoding the design to reduce the number of logic levels to minimize delay, especially along your critical paths.

Default or Others Case Assignment

To fully specify the cases in a CASE statement, include a DEFAULT (Verilog HDL) or OTHERS (VHDL) assignment. This assignment is especially important in one-hot encoding schemes where many combinations of the select lines are unused. Specifying a case for the unused select line combinations gives the synthesis tool information about how to synthesize these cases, and is required by the Verilog HDL and VHDL language specifications.

Some designs do not require that the outcome in the unused cases be considered, often because designers assume these cases will not occur. For these types of designs, you can choose any value for the DEFAULT or OTHERS assignment. However, be aware that the assignment value you choose can have a large effect on the logic utilization required to implement the design due to the different ways synthesis tools treat different values for the assignment, and how the synthesis tools use different speed and area optimizations.

In general, to obtain best results, explicitly define invalid CASE selections with a separate DEFAULT or OTHERS statement instead of combining the invalid cases with one of the defined cases.

If the value in the invalid cases is not important, specify those cases explicitly by assigning the X (don't care) logic value instead of choosing another value. This assignment allows your synthesis tool to perform the best area optimizations.

You can experiment with different DEFAULT or OTHERS assignments for your HDL design and your synthesis tool to test the effect they have on logic utilization in your design.

Implicit Defaults

The IF statements in Verilog HDL and VHDL can be a convenient way to specify conditions that do not easily lend themselves to a CASE-type approach. However, using IF statements can result in complicated multiplexer trees that are not easy for synthesis tools to optimize.

In particular, every IF statement has an implicit ELSE condition, even when it is not specified. These implicit defaults can cause additional complexity in a multiplexed design.

The code in Example 6–47 represents a multiplexer with four inputs (a, b, c, d) and one output (z).

Example 6–47. VHDL IF Statement with Implicit Defaults

```
IF cond1 THEN
   IF cond2 THEN
      z <= a;
   END IF;
ELSIF cond3 THEN
   IF cond4 THEN
      z <= b;
   ELSIF cond5 THEN
      z <= c;
   END IF;
ELSIF cond6 THEN
   z <= d;
END IF;
```

This is not a recommended coding style. Although the code appears to implement a 4:1 multiplexer, each of the three separate IF statements in the code has an implicit ELSE condition that is not specified. Because the output values for the ELSE cases are not specified, the synthesis tool assumes the intent is to maintain the same output value for these cases and infers a combinational loop, such as a latch. Latches add to the design's logic utilization and can also make timing analysis difficult and lead to other problems.

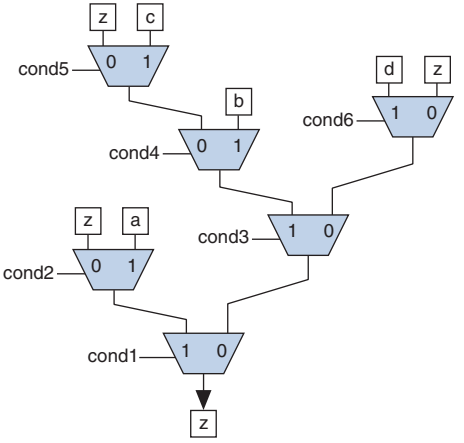
The code sample shown in Example 6–48 shows code with the same functionality as the code shown in Example 6–47, but specifies the ELSE cases explicitly. (This is not a recommended coding style improvement, but it explicitly shows the default conditions from the previous example.)

Example 6–48. VHDL IF Statement with Default Conditions Explicitly Specified

```
IF cond1 THEN
   IF cond2 THEN
      z <= a;
   ELSE
      z <= z;
   END IF;
ELSIF cond3 THEN
   IF cond4 THEN
      z <= b;
   ELSIF cond5 THEN
      z <= c;
   ELSE
      z <= z;
   END IF;
ELSIF cond6 THEN
   z <= d;
ELSE
   z <= z;
END IF;
```

Figure 6–3 is a schematic representing the code in Example 6–48, which illustrates that the multiplexer logic is significantly more complicated than a basic 4:1 multiplexer, although there are only four inputs.

Figure 6–3. Multiplexer Implementation of an IF Statement with Implicit Defaults



There are several ways you can simplify the multiplexed logic and remove the unneeded defaults. The optimal method may be to recode the design so the logic takes the structure of a 4:1 `CASE` statement. Alternatively, if priority is important, you can restructure the code to deduce default cases and flatten the multiplexer. In this example, instead of `IF cond1 THEN IF cond2`, use `IF (cond1 AND cond2)`, which performs the same function. In addition, examine whether the defaults are don't care cases. In this example, you can promote the last `ELSIF cond6` statement to an `ELSE` statement if no other valid cases can occur.

Avoid unnecessary default conditions in your multiplexer logic to reduce the complexity and logic utilization required to implement your design.

Degenerate Multiplexers

A degenerate multiplexer is a multiplexer in which not all of the possible cases are used for unique data inputs. The unneeded cases tend to contribute to inefficiency in the logic utilization for these multiplexers. You can recode degenerate multiplexers so they take advantage of the efficient logic utilization possible with full binary multiplexers.

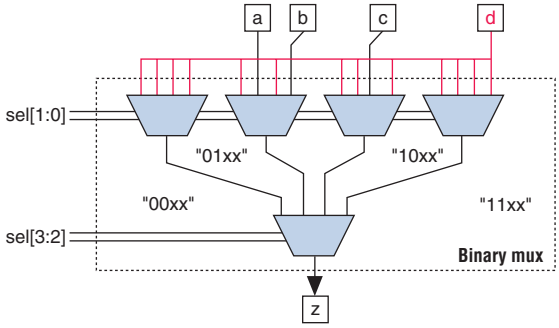
Example 6–49 shows a VHDL CASE statement describing a degenerate multiplexer.

Example 6–49. VHDL CASE Statement Describing a Degenerate Multiplexer

```
CASE sel[3:0] IS
   WHEN "0101" => z <= a;
   WHEN "0111" => z <= b;
   WHEN "1010" => z <= c;
   WHEN OTHERS => z <= d;
END CASE;
```

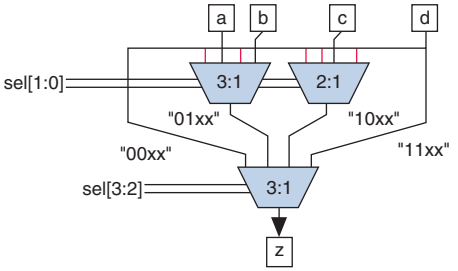
The number of select lines in a binary multiplexer normally dictates the size of a multiplexer needed to implement the desired function. For example, the multiplexer structure represented in Figure 6–4 has four select lines capable of implementing a binary multiplexer with 16 inputs. However, the design does not use all 16 inputs, which makes this multiplexer a degenerate 16:1 multiplexer.

Figure 6–4. Binary Degenerate Multiplexer



In the example in Figure 6–4, the first and fourth multiplexers in the top level can easily be eliminated because all four inputs to each multiplexer are the same value, and the number of inputs to the other multiplexers can be reduced, as shown in Figure 6–5.

Figure 6–5. Optimized Version of the Degenerate Binary Multiplexer



Implementing this version of the multiplexer still requires at least five 4-input LUTs, two for each of the remaining 3:1 multiplexers and one for the 2:1 multiplexer. This design selects an output from only four inputs, a 4:1 binary multiplexer can be implemented optimally in two LUTs, so this degenerate multiplexer tree reduces the efficiency of the logic.

You can improve logic utilization of this structure by recoding the select lines to implement a full 4:1 binary multiplexer. The code sample shown in Example 6–50 shows a recoder design that translates the original select lines into the `z_sel` signal with binary encoding.

Example 6–50. VHDL Recoder Design for Degenerate Binary Multiplexer

```
CASE sel[3:0] IS
   WHEN "0101" => z_sel <= "00";
   WHEN "0111" => z_sel <= "01";
   WHEN "1010" => z_sel <= "10";
   WHEN OTHERS => z_sel <= "11";
END CASE;
```

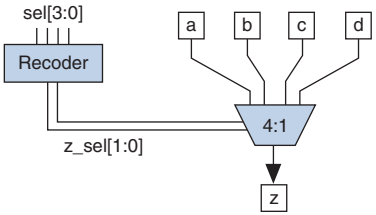
The code sample shown in Example 6–51 shows you how to implement the full binary multiplexer.

Example 6–51. VHDL 4:1 Binary Multiplexer Design

```
CASE z_sel[1:0] IS
   WHEN "00" => z <= a;
   WHEN "01" => z <= b;
   WHEN "10" => z <= c;
   WHEN "11" => z <= d;
END CASE;
```

Use the new z_sel control signal from the recoder design to control the 4:1 binary multiplexer that chooses between the four inputs a, b, c, and d, as illustrated in Figure 6–6. The complexity of the select lines is handled in the recoder design, and the data multiplexing is performed with simple binary select lines enabling the most efficient implementation.

Figure 6–6. Binary Multiplexer with Recorder



The design for the recoder can be implemented in two LUTs and the efficient 4:1 binary multiplexer uses two LUTs, for a total of four LUTs. The original degenerate multiplexer required five LUTs, so the recoded version uses 20% less logic than the original.

You can often improve the logic utilization of multiplexers by recoding the select lines into full binary cases. Although logic is required to perform the encoding, the overall logic utilization is often improved.

Buses of Multiplexers

The inputs to multiplexers are often data input buses in which the same multiplexer function is performed on a set of data input buses. In these cases, any inefficiency in the multiplexer is multiplied by the number of bits in the bus. The issues described in the previous sections become even more important for wide multiplexer buses.

For example, the recoding of select lines into full binary cases detailed in the previous section can often be used in multiplexed buses. Recoding the select lines may need to be completed only once for all the multiplexers in the bus. By sharing the recoder logic among all the bits in the bus, you can greatly improve the logic efficiency of a bus of multiplexers.

The degenerate multiplexer in the previous section requires five LUTs to implement. If the inputs and output are 32 bits wide, the function could require 32 × 5 or 160 LUTs for the whole bus. The recoder design uses only two LUTs, and the select lines only need to be recoded once for the entire bus. The binary 4:1 multiplexer requires two LEs per bit of the bus. The

total logic utilization for the recoded version could be 2 + (2 x 32) or 66 LUTs for the whole bus, compared to 160 LUTs for the original version. The logic savings become more important with wide multiplexer buses.

Using techniques to optimize degenerate multiplexers, removing unneeded implicit defaults, and choosing the optimal `DEFAULT` or `OTHERS` case can play an important role when optimizing buses of multiplexers.

Cyclic Redundancy Check Functions

Cyclic redundancy check (CRC) computations are used heavily by communications protocols and storage devices to detect any corruption of the data. These functions are highly effective; there is a very low probability that corrupted data can pass a 32-bit CRC check.

CRC functions typically use wide XOR gates to compare the data. The way that synthesis tools flatten and factor these XOR gates to implement the logic in FPGA LUTs can greatly impact the area and performance results for the design. XOR gates have a cancellation property which creates an exceptionally large number of reasonable factoring combinations, so synthesis tools cannot always choose the best result by default.

The 6-input ALUT has a significant advantage over 4-input LUTs for these designs. When properly synthesized, CRC processing designs can run at high speeds in devices with 6-input ALUTs.

The following guidelines help you improve the quality of results for CRC designs in Altera devices.

If Performance is Important, Optimize for Speed

Synthesis tools flatten XOR gates to minimize area and depth of levels of logic. Synthesis tools such as Quartus II integrated synthesis target area optimization by default for these logic structures. Therefore, for more focus on depth reduction, set the synthesis optimization technique to speed.



Note that flattening for depth sometimes causes a significant increase in area.

Use Separate CRC Blocks Instead of Cascaded Stages

Some designers optimize their CRC designs to use cascaded stages, for example, four stages of 8 bits. In such designs, intermediate calculations are used as needed (such as the calculations after 8, 24, or 32 bits)

depending on the data width. This design is not optimal in FPGA devices. The XOR cancellations that can be performed in CRC designs mean that the function does not require all the intermediate calculations to determine the final result. Therefore, forcing the use of intermediate calculations increases the area required to implement the function, as well as increasing the logic depth because of the cascading. It is typically better to create full separate CRC blocks for each data width that you need in the design, then multiplex them together to choose the appropriate mode at a given time.

Use Separate CRC Blocks Instead of Allowing Blocks to Merge

Synthesis tools often attempt to optimize CRC designs by sharing resources and extracting duplicates in two different CRC blocks because of the factoring options in the XOR logic. As addressed previously, the CRC logic allows significant reductions but this works best when each CRC function is optimized separately. Check for duplicate extraction behavior if you have different CRC functions that are driven by common data signals or that feed the same destination signals.

If you are having problems with the quality of results and you see that two CRC functions are sharing logic, ensure that the blocks are synthesized independently using one of the following methods:

- Define each CRC block as a separate design partition in an incremental compilation design flow.

  For details, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

- Synthesize each CRC block as a separate project and then write a separate VQM or EDIF netlist file for each.

Take Advantage of Latency if Available

If your design can use more than one cycle to implement the CRC functionality, adding registers and retiming the design can help reduce area, improve performance, and reduce power utilization. If your synthesis tool offers a retiming feature (such as the Quartus II software **Perform gate-level register retiming** option), you can insert an extra bank of registers at the input and allow the retiming feature to move the registers for better results. You can also build the CRC unit half as wide and alternate between halves of the data in each clock cycle.

Save Power by Disabling CRC Blocks When Not in Use

CRC designs are heavy consumers of dynamic power because the logic toggles whenever there is a change in the design. To save power, use clock enables to disable the CRC function for every clock cycle that the logic is not needed. Some designs don't check the CRC results for a few clock cycles while other logic is performed. It is valuable to disable the CRC function even for this short amount of time.

Use the Device Synchronous Load (sload) Signal to Initialize

The data in many CRC designs must be initialized to `1`'s before operation. If your target device supports the use of the `sload` signal, you should use it to set all the registers in your design to `1`'s before operation. To enable use of the `sload` signal, follow the coding guidelines presented in "Secondary Register Control Signals Such as Clear and Clock Enable" on page 6–39. You can check the register equations in the Timing Closure Floorplan or the Chip Planner to ensure that the signal was used as expected.



If you must force a register implementation using an `sload` signal, you can use low-level device primitives as described in the *Introduction to Low-Level Primitives Design User Guide*.

Comparators

Synthesis software, including Quartus II integrated synthesis, uses device and context-specific implementation rules for comparators (`<`, `>`, or `==`) and selects the best one for your design. This section provides some information about the different types of implementations available and provides suggestions on how you can code your design to encourage a specific implementation.

The `==` comparator is implemented in general logic cells. The `<` comparison can be implemented using the carry chain or general logic cells. In devices with 6-input ALUTs, the carry chain is capable of comparing up to three bits per cell. In devices with 4-input LUTs, the capacity is one bit of comparison per cell, similar to an add/subtract chain. The carry chain implementation tends to be faster than the general logic on standalone benchmark test cases, but can result in lower performance when it is part of a larger design due to the increased restriction on the Fitter. The area requirement is similar for most input patterns. The synthesis software selects an appropriate implementation based on the input pattern.

If you are using Quartus II integrated synthesis, you can guide the synthesis by using specific coding styles. To select a carry chain implementation explicitly, rephrase your comparison in terms of addition. As a simple example, the following coding style allows the synthesis tool to select the implementation, which is most likely using general logic cells in modern device families:

```
wire [6:0] a,b;
wire alb = a<b;
```

In the following coding style, the synthesis tool uses a carry chain (except for a few cases, such as when the chain is very short or the signals `a` and `b` minimize to the same signal):

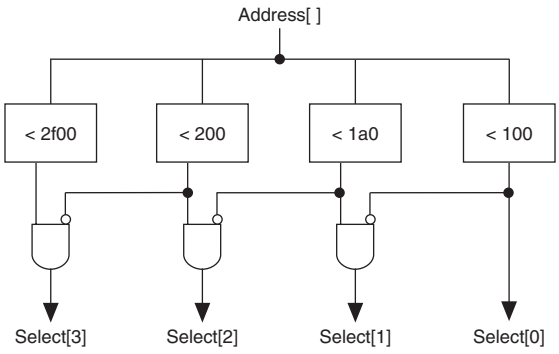
```
wire [6:0] a,b;
wire [7:0] tmp = a - b;
wire alb = tmp[7]
```

This second coding style uses the top bit of the `tmp` signal, which is 1 in twos complement logic if *a* is less than *b*, because the subtraction *a* – *b* results in a negative number.

If you have any information about the range of the input, you have "don't care" values that you can use to optimize the design. Because this information is not available to the synthesis tool, you can often reduce the device area required to implement the comparator with specific hand implementation of the logic.

You can also check whether a bus value is within a constant range with a small amount of logic area by using the logic structure shown in Figure 6–7. This type of logic occurs frequently in address decoders.

Figure 6–7. Example Logic Structure for Using Comparators to Check a Bus Value Range



Counters

Implementing counters in HDL code is easy; they are implemented with an adder followed by registers. Remember that the register control signals, such as enable (ena), synchronous clear (sclr) and synchronous load (sload), are available. For the best area utilization, ensure that the up/down control or controls are expressed in terms of one addition instead of two separate addition operators.

If you use the following coding style, your synthesis tool may implement two separate carry chains for addition (if it doesn't detect the issue and optimize the logic):

```
out <= count_up ? out + 1 : out - 1;
```

The following coding style requires only one adder along with some other logic:

```
out <= out + (count_up ? 1 : -1);
```

In this case, the coding style better matches the device hardware because there is only one carry chain adder, and the –1 constant logic is implemented in the look-up table in front of the adder without adding extra area utilization.

Designing with Low-Level Primitives

Low-level HDL design is the practice of using low-level primitives and assignments to dictate a particular hardware implementation for a piece of logic. Low-level primitives are small architectural building blocks that assist you in creating your design. With the Quartus II software, you can use low-level HDL design techniques to force a specific hardware implementation that can help you achieve better resource utilization or faster timing results.



Using low-level primitives is an advanced technique to help with specific design challenges, and is optional in the Altera design flow. For many designs, synthesizing generic HDL source code and Altera megafunctions gives you the best results.

Low-level primitives allow you to use the following types of coding techniques:

- Instantiate the logic cell or `LCELL` primitive to prevent Quartus II integrated synthesis from performing optimizations across a logic cell
- Create carry and cascade chains using `CARRY`, `CARRY_SUM`, and `CASCADE` primitives

- Instantiate registers with specific control signals using `DFF` primitives
- Specify the creation of LUT functions by identifying the LUT boundaries
- Use I/O buffers to specify I/O standards, current strengths, and other I/O assignments
- Use I/O buffers to specify differential pin names in your HDL code, instead of using the automatically-generated negative pin name for each pair

Refer to the *Designing With Low-Level Primitives User Guide* for details about and examples of using these types of assignments.

Conclusion

Because coding style and megafunction implementation can have such a large effect on your design performance, it is important to match the coding style to the device architecture from the very beginning of the design process. To improve design performance and area utilization, take advantage of advanced device features, such as memory and DSP blocks, as well as the logic architecture of the targeted Altera device by following the coding recommendations presented in this chapter.



For additional optimization recommendations, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Referenced Documents

This chapter references the following documents:

- *Area and Timing Optimization* in volume 2 of the *Quartus II Handbook*
- *Design Recommendations for Altera Devices* in volume 1 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* in volume 1 of the *Quartus II Handbook*
- *Synthesis* section in volume 1 of the *Quartus II Handbook*

Document Revision History

Table 6–2 shows the revision history for this chapter.

Table 6–2. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 6–74.	Updates for the Quartus II software version 7.2.
May 2007 v7.1.0	Updates for the Quartus II software version 7.1 release, including: • Added Quartus II Language Templates. • Updated text in Using Altera Megafunctions. • Updated Table 6-1. • Added Avoide Unsupported Reset Conditions. • Added Check Read-During-Write Behavior. • Added True Dual-Port Synchronous RAM. • Added Specifying Initial Memory Contents at Power-Up. • Added Referenced Documents.	Updates for the Quartus II software version 7.1, including the addition of Arria GX devices, new HDL design templates, new support for inferring true dual-port RAM blocks. Clarified RAM inference guidelines with respect to synchronous memory and read-during-write behavior.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Updates for the Quartus II software version 6.1 release, including: • Moved the "Simple Dual-Port, Dual-Clock Synchronous RAM" on page 7–19 section within the chapter • Added information about read-through-write conditions • Added example code, including Examples 7–13 and 7–14; Examples 7–17 and 7–19; and Example 7–23 • Added a section about "Designing with Low-Level Primitives" on page 7–71 • Added information about implementing a safe state machine • Reorganized the chapter, shuffling the "Coding Guidelines for Registers and Latches" and "General Coding Guidelines" and the subsections therein • Added "Comparators" on page 7–69 and "Counters" on page 7–71 to the General Coding Guidelines section	Updates for the Quartus II software version 6.1, including the addition of Stratix III devices. Changes to the recommendations for RAM block inference to ensure better quality of results, and new suggestions for different general logic structures.
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—

Table 6–2. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
May 2005 v5.0.0	Chapter 4 was formerly Chapter 1 in version 4.2.	—
December 2004 v2.1	Updated for Quartus II software version 4.2: • Chapter 4 was formerly Chapter 1. • General formatting and editing updates. • Device family support descriptions updated. • Updated HardCopy structured support for performance improvements. • Quartus II Archive File automatically receives buffer insertion. • Power Calculator now Power Estimator for affected devices. • Updates to tables, figures. • The description of How to Design HardCopy Stratix Devices was updated. • The description of HardCopy Timing Optimization Wizard was updated. • *HardCopy Floorplans and Timing Modules* was renamed to *Design Optimization*. • The description of Performance Estimation was updated. • Added new section on Buffer Insertion. • Location Constraints was updated. • Targeting Designs to HardCopy APEX 20KC and HardCopy APEX 20KE Devices was removed. • A new section *Altera Recommended HDL* was added. • Table 2–5 was added. It lists the HardCopy Stratix design files collected by the hardCopy Files Wizard. • The description of the HardCopy APEX Power Estimator was updated. • A new section about Targeting Designs to HardCopy APEX Devices was added.	—

Section III. Synthesis

As programmable logic devices (PLDs) become more complex and require increased performance, advanced design synthesis has become an important part of the design flow. In the Quartus® II software you can use the Analysis and Synthesis module of the Compiler to analyze your design files and create the project database. You can also use other EDA synthesis tools to synthesize your designs, and then generate EDIF netlist files or VQM files that can be used with the Quartus II software. This section explains the options that are available for each of these flows, and how they are supported in the Quartus II, version 7.2 software.

This section includes the following chapters:

- Chapter 7, Synplicity Synplify and Synplify Pro Support
- Chapter 8, Quartus II Integrated Synthesis
- Chapter 9, Mentor Graphics LeonardoSpectrum Support
- Chapter 10, Mentor Graphics Precision RTL Synthesis Support
- Chapter 11, Synopsys Design Compiler FPGA Support
- Chapter 12, Analyzing Designs with Quartus II Netlist Viewers



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

7. Synplicity Synplify and Synplify Pro Support

QII51009-7.2.0

Introduction

As programmable logic device (PLD) designs become more complex and require increased performance, advanced synthesis has become an important part of the design flow. This chapter documents support for the Synplicity Synplify and Synplify Pro software in the Quartus® II software, as well as key design flows, methodologies, and techniques for achieving good results in Altera® devices. This chapter includes the following topics:

- General design flow with the Synplify and Quartus II software
- Synplify software optimization strategies, including timing-driven compilation settings, optimization options, and Altera-specific attributes
- Exporting designs and constraints to the Quartus II software using NativeLink® integration
- Guidelines for Altera megafunctions and library of parameterized module (LPM) functions, instantiating them with the MegaWizard® Plug-In Manager, and tips for inferring them from hardware description language (HDL) code
- Incremental compilation and block-based design, including the Synplify Pro software MultiPoint flow

The content in this chapter applies to both the Synplify and Synplify Pro software unless otherwise specified.This chapter includes the following sections:

- "Altera Device Family Support" on page 7–2
- "Design Flow" on page 7–3
- "Synplify Optimization Strategies" on page 7–8
- "Exporting Designs to the Quartus II Software Using NativeLink Integration" on page 7–17
- "Guidelines for Altera Megafunctions and Architecture-Specific Features" on page 7–32
- "Incremental Compilation and Block-Based Design" on page 7–44

This chapter assumes that you have set up, licensed, and are familiar with the Synplify or Synplify Pro software.

Altera Device Family Support

The Synplify software maps synthesis results to Altera device families. The following list shows the Altera device families supported by the Synplify software version 9.0, with the Quartus II software version 7.2:

- Cyclone® III
- Stratix® III
- Stratix II, Stratix II GX, Hardcopy® II
- Stratix, Stratix GX, HardCopy Stratix
- Cyclone II
- Cyclone
- MAX® II
- MAX® 7000, MAX 3000
- APEX™ II
- APEX 20K, APEX 20KC, APEX 20KE
- FLEX® 10K, FLEX 6000
- ACEX® 1K

The Synplify software also supports the following legacy devices that are supported in the Quartus II software only with a specific license requested at **www.altera.com/mysupport**:

- Excalibur™ ARM®
- Mercury™

The Synplify software also supports the following legacy devices that are supported only in the Altera MAX+PLUS II software:

- FLEX 8000
- MAX 9000



To learn about new device support for a specific Synplify version, refer to the release notes on Synplicity's web site at **www.synplicity.com.**

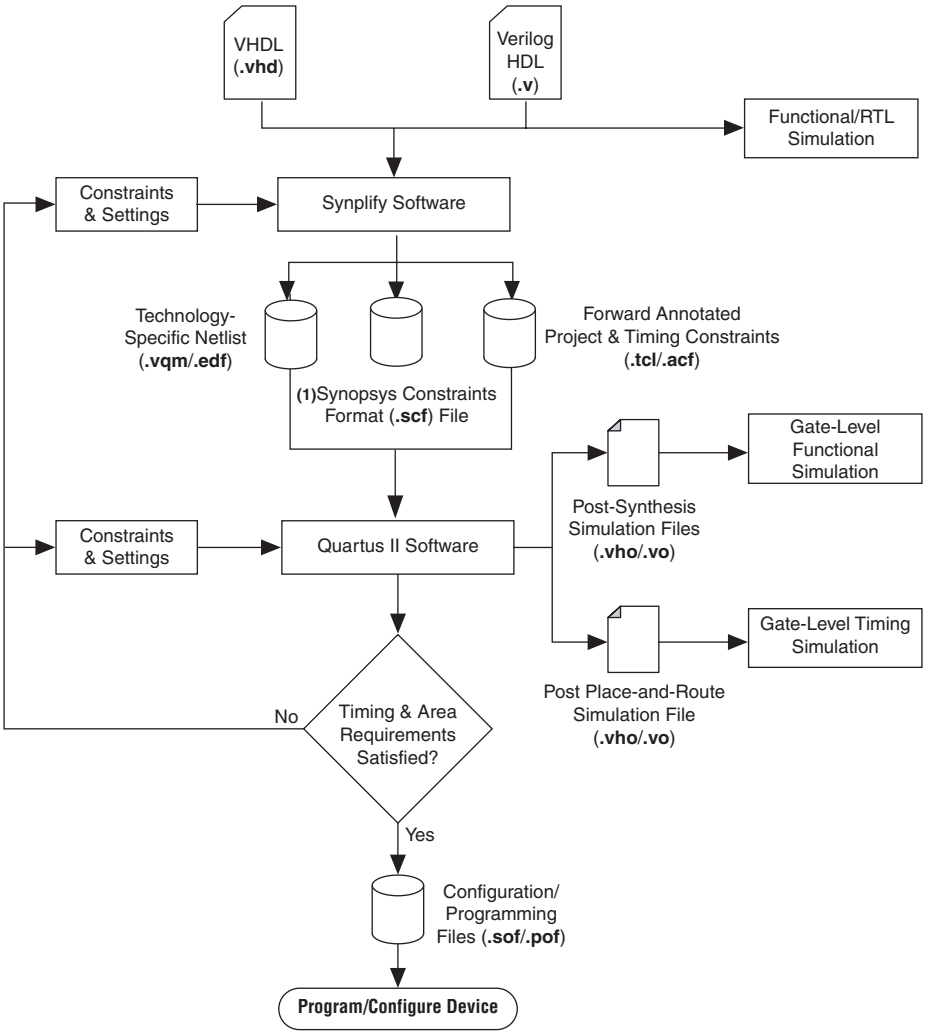
Design Flow

A Quartus II software design flow using the Synplify software consists of the following steps:

1. Create Verilog HDL or VHDL design files in the Quartus II software, Synplify software, or a text editor.
2. Set up a project in the Synplify software and add the HDL design files for synthesis.
3. Select a target device and add timing constraints and compiler directives to optimize the design during synthesis.
4. Create a Quartus II project and import the technology-specific EDIF (**.edf**) or VQM (**.vqm**) netlist, the Synopsys Constraints Format (**.scf**) file (for TimeQuest constraints if a Stratix III or Cyclone III device is selected), and the tool command language (**.tcl)** constraint file generated by the Synplify software into the Quartus II software for placement and routing, and for performance evaluation.
5. After obtaining place-and-route results that meet your needs, configure or program the Altera device.

Figure 7–1 shows the recommended design flow when using the Synplify and the Quartus II software.

Figure 7–1. Recommended Design Flow



The Synplify and Synplify Pro software support both VHDL and Verilog HDL source files. The Synplify Pro software also supports mixed synthesis, allowing a combination of VHDL and Verilog HDL source files.

Specify timing constraints and attributes for the design in a Synplify Constraints File (**.sdc**) with the SCOPE window in the Synplify software or directly in the HDL source file. Compiler directives can also be defined in the HDL source file. Many of these constraints are forward-annotated for use by the Quartus II software.

The HDL Analyst that is included in the Synplify software is a graphical tool for generating schematic views of the technology-independent register transfer level (RTL) view netlist (**.srs**) and technology-view netlist (**.srm**) files. You can use the Synplify HDL Analyst to analyze and debug your design visually. The HDL Analyst supports cross probing between the RTL and Technology views, the HDL source code, and the Finite State Machine (FSM) viewer. Refer to "FSM Compiler" on page 7–11.



A separate license file is required to enable the HDL Analyst in the Synplify software. The Synplify Pro software includes the HDL Analyst.

Once synthesis is complete, import the electronic design interchange format (EDIF) or Verilog Quartus Mapping (VQM) netlist to the Quartus II software for place-and-route. You can use the Tcl file generated by the Synplify software to forward-annotate your constraints (including device selection), and optionally to set up your project in the Quartus II software.

If a Stratix III or Cyclone III device is selected, the Quartus II software uses the SDC-format timing constraints from the **.scf** file with the TimeQuest Timing Analyzer by default. For other devices, the Quartus II software uses the Tcl Classic Timing Analyzer timing constraints written to the Quartus Setting File (**.qsf**). Refer to "Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File" on page 7–20 for information about to manually changing from the TimeQuest Timing Analyzer to the Classic Timing Analyzer manually for Stratix III and Cyclone III devices.

If the area and timing requirements are satisfied, use the files generated by the Quartus II software to program or configure the Altera device. As shown in Figure 7–1, if your area or timing requirements are not met, you can change the constraints in the Synplify software or the Quartus II software and repeat the synthesis. Repeat the process until the area and timing requirements are met.

While you can perform simulation at various points in the process, final timing analysis should be performed after placement and routing is complete. Formal verification may also be performed at various stages of the design process.

For more information about how the Synplify software supports formal verification, refer to the *Formal Verification* section in volume 3 of the *Quartus II Handbook*.

You can also use other options and techniques in the Quartus II software to meet area and timing requirements. One such option is called WYSIWYG Primitive Resynthesis, which can perform optimizations on your VQM netlist within the Quartus II software.

For information about netlist optimizations, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

In some cases, you may be required to modify the source code if area and timing requirements cannot be met using options in the Synplify and Quartus II software.

After synthesis, the Synplify software produces several intermediate and output files. Table 7–1 lists these file types.

Table 7–1. Synplify Intermediate and Output Files (Part 1 of 2)

File Extensions	File Description
.srs	Technology-independent RTL netlist that can be read only by the Synplify software
.srm	Technology view netlist
.srr *(1)*	Synthesis Report file
.edf/.vqm *(2)*	Technology-specific netlist in electronic design interchange format (EDIF) or VQM file format

Table 7–1. Synplify Intermediate and Output Files (Part 2 of 2)

File Extensions	File Description
.acf/.tcl *(3)*	Forward-annotated constraints file containing constraints and assignments
.scf	Synopsys Constraint Format file containing timing constraints for the TimeQuest Timing Analyzer

Notes to Table 7–1:

(1) This report file includes performance estimates that are often based on pre-place-and-route information. Use the f_{MAX} reported by the Quartus II software after place-and-route—it is the only reliable source of timing information. This report file includes post-synthesis device resource utilization statistics that may inaccurately predict resource usage after place-and-route. The Synplify software does not account for black box functions nor for logic usage reduction achieved through register packing performed by the Quartus II software. Register packing combines a single register and look-up table (LUT) into a single logic cell, reducing the logic cell utilization below the Synplify software estimate. Use the device utilization reported by the Quartus II software after place-and-route.

(2) An EDIF output file (**.edf**) is created for ACEX 1K, FLEX 10K, FLEX 10KA, FLEX 10KE, FLEX 6000, FLEX 8000, MAX 7000, MAX 9000, and MAX 3000 devices. A VQM file is created for all other Altera device families.

(3) An Assignment and Configuration File (**.acf**) file is created only for ACEX 1K, FLEX 10K, FLEX 10KA, FLEX 10KE, FLEX 6000, FLEX 8000, MAX 7000, MAX 9000, and MAX 3000 devices. The ACF is generated for backward compatibility with the MAX+PLUSII software. A Tcl file for the Quartus II software is created for all devices. The Tcl file contains the appropriate Tcl commands to create and set up a Quartus II project and, if applicable, the MAX+PLUS II assignments are imported from the ACF file.

Output Netlist File Name and Result Format

Specify the output netlist directory location and name by performing the following steps:

1. On the Project menu, click **Implementation Options**.
2. Click the **Implementation Results** tab.
3. In the **Results Directory** box, type your output netlist file directory location.
4. In the **Result File Name** box, type your output netlist file name.

By default, directory and file name are set to the project implementation directory and the top-level design module or entity name.

The **Result Format** and **Quartus version** options are also available on the **Implementation Results** tab. The **Result Format** list specifies an EDIF or VQM netlist depending on your device family. The software creates an EDIF output netlist file only for ACEX 1K, FLEX 10K, FLEX 6000, FLEX 8000, MAX 7000, MAX 9000, and MAX 3000 devices. For other Altera devices, the software generates a VQM-formatted netlist.

Beginning with the Synplify software version 8.4, select the version of the Quartus II software that you are using in the **Quartus version** list. This option ensures that the netlist is compatible with the software version and supports the newest features. Altera recommends using the latest version of the Quartus II software whenever possible. If your Quartus II software is newer than the versions available in the **Quartus version** list, check if there is a newer version of the Synplify software available that supports the current Quartus II software version. Otherwise, choose the latest version in the list for the best compatibility.



The Quartus version list is available only after selecting an Altera device.

Synplify Optimization Strategies

As designs become more complex and require increased performance, using different optimization strategies has become important. Combining Synplify software constraints with VHDL and Verilog HDL coding techniques and Quartus II software options can help you obtain the required results.



For additional design and optimization techniques, refer to the *Design Recommendations for Altera Devices* chapter in volume 1 and the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

The Synplify software offers many constraints and optimization techniques to improve your design's performance. The Synplify Pro software adds some additional techniques that are not supported in the basic Synplify software. Wherever this document describes Synplify support, this includes both the basic Synplify and the Synplify Pro software; Synplify Pro-only features are labeled as such. This section provides an overview of some of the techniques you can use to help improve the quality of your results.



For more information about applying the attributes discussed in this section, refer to the *Tasks and Tips* chapter of the *Synplify Software User Guide*.

Implementations in Synplify Pro

To create different synthesis results without overwriting the others, In the Synplify Pro software, on the Project menu, click **New Implementation**. For each implementation, specify the target device, synthesis options, and constraint files. Each implementation generates its own subdirectory that contains all the resulting files, including VQM/EDIF, **.scf** and Tcl files, from a compilation of the particular implementation. You can then compare the results of the different implementations to find the optimal set of synthesis options and constraints for a design.

Timing-Driven Synthesis Settings

The Synplify software supports timing-driven synthesis with user-assigned timing constraints to optimize the performance of the design. The Synplify software optimizes the design to attempt to meet these constraints.

The Quartus II NativeLink feature allows timing constraints that are applied in the Synplify software to be forward-annotated for the Quartus II software using either a Tcl script file or a **.scf** file for timing-driven place and route. Refer to "Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File" on page 7–20 or "Passing Constraints to the Quartus II Software using Tcl Commands" on page 7–22 for more details about how constraints such as clock frequencies, false paths, and multicycle paths are forward-annotated. This section explains some of the important timing constraints in the Synplify software.



The Synplify Synthesis Report File (**.srr**) contains timing reports of estimated place-and-route delays. The Quartus II software can perform further optimizations on a post-synthesis netlist from third-party synthesis tools. In addition, designs may contain black boxes or intellectual property (IP) functions that have not been optimized by the third-party synthesis software. Actual timing results are obtained only after the design has gone through full placement and routing in the Quartus II software. For these reasons, the Quartus II post place-and-route timing reports provide a more accurate representation of the design. The statistics in these reports should be used to evaluate design performance.

Clock Frequencies

For single-clock designs, specify a global frequency when using the push-button flow. While this flow is simple and provides good results, often it does not meet the performance requirements for more advanced designs. You can use timing constraints, compiler directives, and other attributes to help optimize the performance of a design. You can enter these attributes and directives directly in the HDL code. Alternatively, you can enter attributes (not directives) into an **.sdc** file with the SCOPE window in the Synplify software.

Use the SCOPE window to set global frequency requirements for the entire design and individual clock settings. Use the **Clocks** tab in the SCOPE window to specify frequency (or period), rise times, fall times, duty cycle, and other settings. Assigning individual clock settings, rather than over-constraining the global frequency, helps the Quartus II

software and the Synplify software achieve the fastest clock frequency for the overall design. The `define_clock` attribute assigns clock constraints.

Multiple Clock Domains

The Synplify software can perform timing analysis on unrelated clock domains. Each clock group is a different clock domain and is treated as unrelated to the clocks in all other clock groups. All the clocks in a single clock group are assumed to be related and the Synplify software automatically calculates the relationship between the clocks. You can assign clocks to a new clock group, or put related clocks in the same clock group by using the **Clocks** tab in the SCOPE window or with the `define_clock` attribute.

Input/Output Delays

Specify the input and output delays for the ports of a design in the **Input/Output** tab of the SCOPE window or with the `define_input_delay` and `define_output_delay` attributes. The Synplify software does not allow you to assign the t_{CO} and t_{SU} values directly to inputs and outputs. However, a t_{CO} value can be inferred by setting an external output delay, and a t_{SU} value can be inferred by setting an external input delay. Equation 1 and 2 below illustrate the relationship between t_{CO} / t_{SU} and the input/output delays:

(1) t_{CO} = *clock period – external output delay*

(2) t_{SU} = *clock period – external input delay*

When the `syn_forward_io_constraints` attribute is set to 1, the Synplify software passes the external input and output delays to the Quartus II software using NativeLink integration. The Quartus II software then uses the external delays to calculate the maximum system frequency.

Multicycle Paths

Specify any multicycle paths in the design in the **Multi-Cycle Paths** tab of the SCOPE window or with the `define_multicycle_path` attribute. A multicycle path is a path that requires more than one clock cycle to propagate. It is important to specify which paths are multicycle to avoid having the Quartus II and the Synplify compilers work excessively on a non-critical path. Not specifying these paths can also result in an inaccurate critical path being reported during timing analysis.

False Paths

False paths are paths that should not be considered during timing analysis or which should be assigned low (or no) priority during optimization. Some examples of false paths are slow asynchronous resets and test logic added to the design. Set these paths in the **False Paths** tab of the SCOPE window. Use the `define_false_path` attribute.

FSM Compiler

If the FSM Compiler is turned on, the compiler automatically detects state machines in a design. The compiler can then extract and optimize the state machine. The FSM Compiler analyzes the state machine and decides to implement sequential, gray, or one-hot encoding based on the number of states. It also performs unused-state analysis, optimization of unreachable states, and minimization of transition logic.

If the FSM Compiler is turned off, the compiler does not infer state machines. The state machines are implemented as coded in the HDL code. Thus, if the coding style for the state machine was sequential, then the implementation is also sequential. If the FSM Compiler is turned on, the compiler infers the state machines. The implementation is based on the number of states regardless of the coding style in the HDL code.

You can use the `syn_state_machine` complier directive to specify or prevent a state machine from being extracted and optimized. To override the default encoding of the FSM Compiler, use the `syn_encoding` directive.

The values for the `syn_encoding` directive are shown in Table 7–2.

Table 7–2. syn_encoding Directive Values

Value	Description
Sequential	Generates state machines with the fewest possible flip-flops. Sequential, also called binary, state machines are useful for area-critical designs when timing is not the primary concern.
Gray	Generates state machines where only one flip-flop changes during each transition. Gray-encoded state machines tend to be free of glitches.
One-hot	Generates state machines containing one flip-flop for each state. One-hot state machines typically provide the best performance and shortest clock-to-output delays. However, one-hot implementations are usually larger than binary implementations.
Safe	Generates extra control logic to force the state machine to the reset state if an invalid state is reached. The safe value can be used in conjunction with the other three values, which results in the state machine being implemented with the requested encoding scheme and the generation of the reset logic.

Example 7–1 shows sample VHDL code for applying the syn_encoding directive.

Example 7–1. VHDL Code for syn_encoding

```
SIGNAL current_state : STD_LOGIC_VECTOR(7 DOWNTO 0);
ATTRIBUTE syn_encoding : STRING;
ATTRIBUTE syn_encoding OF current_state : SIGNAL IS "sequential";
```

The default is to optimize state machine logic for speed and area, but this is potentially undesirable for critical systems. The safe value generates extra control logic to force the state machine to the reset state if an invalid state is reached.

FSM Explorer in Synplify Pro

The Synplify Pro software can use the FSM Explorer to explore different encoding styles for a state machine automatically, and then implement the best encoding based on the overall design constraints. The FSM Explorer uses the FSM Compiler to identify and extract state machines from a design. However, unlike the FSM Compiler which chooses the encoding style based on the number of states, the FSM Explorer tries several different encoding styles before choosing a specific one. The trade-off is that the compilation requires more time to perform the analysis of the state machine, but finds an optimal encoding scheme for the state machine.

Optimization Attributes and Options

The following sections describe other attributes and options that you can modify in the Synplify software to improve your design performance.

Retiming in Synplify Pro

The Synplify Pro software can retime a design, which can improve the timing performance of sequential circuits by moving registers (register balancing) across combinational elements. Be aware that retimed registers incur name changes. To retime your design, turn on the **Retiming** option in the **Device** tab in the **Implementation Options** section, or use the syn_allow_retiming attribute.

Maximum Fan-Out

When your design has critical path nets with high fan-out, you can use the syn_maxfan attribute to control the fan-out of the net. Setting this attribute for a specific net results in the replication of the driver of the net to reduce the overall fan-out. The syn_maxfan attribute takes an integer

value and applies it to inputs or registers. (The syn_maxfan attribute cannot be used to duplicate control signals, and the minimum allowed value of the attribute is 4.) Using this attribute may result in increased logic resource utilization, thus putting a strain on routing resources and leading to long compile times and difficult fitting.

If you need to duplicate an output register or output enable register, you can create a register for each output pin by using the syn_useioff attribute (refer to "Register Packing").

Preserving Nets

During synthesis, the compiler maintains ports, registers, and instantiated components. However, some nets may not be maintained to create an optimized circuit. Applying the syn_keep directive overrides the optimization of the compiler and preserves the net during synthesis. The syn_keep directive takes a Boolean value and can be applied to wires (Verilog HDL) and signals (VHDL). Setting the value to **true** preserves the net through synthesis.

Register Packing

Altera devices allow for the packing of registers into I/O cells. Altera recommends allowing the Quartus II software to make the I/O register assignments. However, it is possible to control register packing with the syn_useioff attribute. The syn_useioff attribute takes a Boolean value and can be applied to ports or entire modules. Setting the value to 1 instructs the compiler to pack the register into an I/O cell. Setting the value to 0 prevents register packing in both the Synplify and Quartus II software.

Resource Sharing

The Synplify software uses resource sharing techniques during synthesis by default to reduce area. Turning off the **Resource Sharing** option on the **Options** tab of the **Implementation Options** dialog box can improve performance results for some designs. If you turn off this option, be sure to check the results to determine if it helps the timing performance; if it does not help, then you should leave **Resource Sharing** turned on.

Preserving Hierarchy

The Synplify software performs cross-boundary optimization by default. This results in the flattening of the design to allow optimization. Use the syn_hier attribute to over-ride the default compiler settings. The

syn_hier attribute takes a string value and applies it to modules and/or architectures. Setting the value to **hard** maintains the boundaries of a module and/or architecture, and prevents cross-boundary optimization.

By default, the Synplify software generates a hierarchical VQM file. To flatten the file, set the syn_netlist_hierarchy attribute equal to 0.

Register Input and Output Delays

The advanced options called define_reg_input_delay and define_reg_output_delay can speed up paths feeding a register or coming from a register by a specific number of nanoseconds. The Synplify software attempts to meet the global clock frequency goals for a design as well as the individual clock frequency goals (set with define_clock). You can use these attributes to add delay to paths feeding into or out of registers to further constrain critical paths.

These options are useful to close timing when your design does not meet timing goals because the routing delay after placement and routing exceeds the delay predicted by the Synplify software. Rerun synthesis using this option, specifying the actual routing delay (from place-and-route results) so that the tool can meet the required clock frequency.

In the SCOPE constraint window, use the registers panel with the following entries:

- **Register**—Specifies the name of the register. If you have initialized a compiled design, you can choose the name from the list.
- **Type**—Specifies whether the delay is an input or output delay.
- **Route**—Shrinks the effective period for the constrained registers by the specified value without affecting the clock period that is forward-annotated to the Quartus II software.

Use the following Tcl command syntax to specify an input or output register delay in nanoseconds.

Example 7–2. Specifying an Input or Output Register Delay Using Tcl Command Syntax

```
define_reg_input_delay {<register>}  -route <delay in ns>
define_reg_output_delay {<register>} -route <delay in ns>
```

syn_direct_enable

This attribute controls the assignment of a clock-enable net to the dedicated enable pin of a register. Using this attribute, you can direct the Synplify mapper to use a particular net as the only clock enable when the design has multiple clock enable candidates.

You can also use this attribute as a compiler directive to infer registers with clock enables. To do so, enter the `syn_direct_enable` directive in your source code, not the SCOPE spreadsheet.

The `syn_direct_enable` data type is Boolean. A value of **1** or **true** enables net assignment to the clock-enable pin. The syntax for Verilog HDL is shown below:

```
object /* synthesis syn_direct_enable = 1 */ ;
```

Standard I/O Pad

For certain Altera devices and the equivalent device I/O standard, you can specify the I/O standard type to use for the I/O pad in the design using the **I/O Standard** panel in the Synplify SCOPE window.

Example 7–3 shows the Synplify SDC syntax for the **define_io_standard** constraint, in which the `delay_type` must be either `input_delay` or `output_delay`.

Example 7–3. Synplify SDC Syntax for the define_io_standard Constraint

```
define_io_standard [-disable|-enable] {<objectName>} -delay_type \
[input_delay|output_delay] <columnTclName>{<value>} \
[<columnTclName>{<value>}...]
```



For details about supported I/O standards, refer to *Altera I/O Standards* in the *Synplify Reference Manual*.

Altera-Specific Attributes

The following attributes are for use with specific Altera device features. These attributes are forward-annotated to the Quartus II project and are used during the place-and-route process.

altera_chip_pin_lc

Use this attribute to make pin assignments. This attribute takes a string value and applies it to inputs and outputs. The attribute can be used only on the ports of the top-level entity in the design, and cannot be used to assign pin locations from entities at lower levels of the design hierarchy.



This attribute is not supported for any of the MAX series devices. In the SCOPE window, select the attribute **altera_chip_pin_lc** and set the value to a pin number or a list of pin numbers.

Example 7–4 shows VHDL code for making location assignments to ACEX 1K and FLEX 10KE devices.



The "@" is used to specify pin locations for ACEX 1K and FLEX 10KE devices. For these devices, the pin location assignments are written to the output EDIF.

Example 7–4. Making Location Assignments to ACEX 1K and FLEX 10KE Devices, VHDL

```
ENTITY sample (data_in : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
   data_out: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
ATTRIBUTE altera_chip_pin_lc : STRING;
ATTRIBUTE altera_chip_pin_lc OF data_out : SIGNAL IS "@14, @5,@16, @15";
```

Example 7–5 shows VHDL code for making location assignments for other Altera devices. The pin location assignments for these devices are written to the output Tcl script.

Example 7–5. Making Location Assignments to Other Devices, VHDL

```
ENTITY sample (data_in : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
   data_out: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
ATTRIBUTE altera_chip_pin_lc : STRING;
ATTRIBUTE altera_chip_pin_lc OF data_out : SIGNAL IS "14, 5, 16,
   15";
```



The `data_out` signal is a 4-bit signal; `data_out[3]` is assigned to pin 14 and `data_out[0]` is assigned to pin 15.

altera_implement_in_esb or altera_implement_in_eab

You can use these attributes to implement logic in either embedded system blocks (ESBs) or embedded array blocks (EABs) rather than in logic resources to improve area utilization. The modules selected for such implementation cannot have feedback paths, and either all or none of the

I/Os must be registered. This attribute takes a boolean value and can be applied to instances. (This option is applicable for devices with ESBs/EABs only. For example, the Stratix family of devices is not supported by this option. This attribute is ignored for designs targeting devices that do not have ESBs or EABs.)

altera_io_powerup

You can use this attribute to define the power-up value of an I/O register that has no set or reset. This attribute takes a string value (high | low) and applies it to ports that have I/O registers.

altera_io_opendrain

Use this attribute to specify open-drain mode I/O ports. This attribute takes a boolean value and applies it to outputs or bidirectional ports for devices that support open-drain mode.

Exporting Designs to the Quartus II Software Using NativeLink Integration

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools, and allows you to run other EDA design entry or synthesis, simulation, and timing analysis tools automatically from within the Quartus II software. After a design is synthesized in the Synplify software, a VQM (or EDIF) netlist file, an **.scf** file for TimeQuest Timing Analyzer timing constraints, and Tcl files are used to import the design into the Quartus II software for place-and-route. You can run the Quartus II software from within the Synplify software or as a standalone application. Once you have imported the design into the Quartus II software, you can specify different options to further optimize the design.



When you are using NativeLink integration, the path to your project must not contain white space. The Synplify software uses Tcl scripts to communicate with the Quartus II software, and the Tcl language does not accept arguments with white space in the path.

You can use NativeLink integration to integrate the Synplify software and Quartus II software with a single GUI for both the synthesis and place-and-route operations. NativeLink integration allows you to run the Quartus II software from within the Synplify software GUI or to run the Synplify software from within the Quartus II software GUI.

This section explains the different Nativelink flows and provides details on how constraints are passed to the Quartus II software. This section describes the following topics:

- "Running the Quartus II Software from within the Synplify Software" on page 7–18
- "Using the Quartus II Software to Run the Synplify Software" on page 7–19
- "Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script" on page 7–19
- "Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File" on page 7–20
- "Passing Constraints to the Quartus II Software using Tcl Commands" on page 7–22

Running the Quartus II Software from within the Synplify Software

To use the Quartus II software from within the Synplify software, you must first verify that the `QUARTUS_ROOTDIR` environment variable contains the Quartus II software installation directory. This environment variable is required to use the Synplify and Quartus II software together.

Under each Implementation in the Synplify Pro software, you can create a place-and-route implementation called **pr_**<*number*> **Altera Place and Route**. You can create new place and route implementations using the **New P&R** button in the GUI. To run the Quartus II software in command-line mode after each synthesis run, use the text box to turn on the place-and-route implementation. The results of the place and route are written to a log file in the **pr_**<*number*> directory under the current implementation directory.

You can also use the commands in the Quartus II menu to run the Quartus II software at any time following a successful completion of synthesis. Use one of the following commands from the **Quartus II** submenu under the Options menu in the Synplify software:

- **Launch Quartus**—Opens the Quartus II software GUI and creates a Quartus II project with the synthesized output file, forward-annotated timing constraints, and pin assignments. You can use this to configure options for the project and execute any Quartus II commands.
- **Run Background Compile**—Runs the Quartus II software in command-line mode with the project settings from the synthesis run. The results of the place-and-route are written to a log file.

The <*project_name*>**_cons.tcl** file is used to set up the Quartus II project and calls the <*project_name*>**.tcl** file to pass constraints from the Synplify software to the Quartus II software. By default, the <*project_name*>**.tcl** file contains device, timing, and location assignments. If a Stratix III or a Cyclone III device is selected, the <*project_name*>**.tcl** file contains the command to use the Synplify-generated **.scf** constraints file with the TimeQuest Timing Analyzer instead of using the Tcl constraints with the Classic Timing Analyzer.

Using the Quartus II Software to Run the Synplify Software

You can set up the Quartus II software to run the Synplify software for synthesis using NativeLink integration. This feature allows you to use the Synplify software to synthesize a design as part of a normal compilation in the Quartus II software.

To set up Synplify in Quartus II, on the Tools menu, click **Options**. In the **Options** window, click **EDA Tool Options** and specify the path of Synplify or Synplify Pro software.



For detailed information about using NativeLink integration with the Synplify software, refer to the Quartus II Help.



If you are running the Quartus II software version 7.1 or later, running the Synplify software with NativeLink integration is supported on both floating network and node-locked single-PC licenses. Both types of licenses support batch mode compilation.

Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script

You can also use the Quartus II software separately from the Synplify software. To run the Tcl script generated by the Synplify software to set up your project and set up assignments such as the device selection, perform the following steps:

1. Ensure the VQM/EDIF, .**scf** (if you are using the TimeQuest Timing Analyzer timing constraints), and Tcl files are located in the same directory (they should be located in the implementation directory by default).

2. In the Quartus II software, on the View menu, point to **Utility Windows** and click **Tcl Console**. The Quartus II Tcl Console opens.

3. At the Tcl Console command prompt, type the following:

```
source <path>/<project name>_cons.tcl ↵
```

Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File

The TimeQuest Timing Analyzer is a powerful ASIC-style timing analysis tool that validates the timing performance of all logic in your design using an industry standard constraints format, Synopsys Design Constraints (SDC). This section explains how timing constraints set in Synplify are passed to the Quartus II software for use with the TimeQuest Timing Analyzer.

The timing constraints you set in Synplify are stored in the Synplify Design Constraints (**.sdc**) file. The Tcl file always contains all other constraints for the Quartus II software, such as the device specification and any location constraints. The timing constraints are forward-annotated using the Tcl file for the Quartus II Classic Timing Analyzer, as described in "Passing Constraints to the Quartus II Software using Tcl Commands" on page 7–22. For the TimeQuest Timing Analyzer, the timing constraints are forward-annotated in the Synopsys Constraints Format (**.scf**) file.

Altera recommends that you use the TimeQuest Timing Analyzer for Stratix III and Cyclone III devices, as specified in the Synplify Tcl file that sets up the Quartus II project. However, you can continue to use the Tcl commands for the Classic Timing Analyzer if required. You can manually change from the TimeQuest Timing Analyzer to the Classic Timing Analyzer in the Quartus II software by performing the following steps:

1. From the Assignments menu, select **Settings**.
2. Click **Timing Analysis Settings.**
3. Under **Timing analysis processing,** click the **Use Classic Timing Analyzer during compilation** radio button. Click **OK**.



For addition information about the TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in the *Quartus II Handbook*.

Synplicity recommends that you modify constraints using the SCOPE constraint editor window and not through the generated **.sdc**, **.scf** or **.tcl** file.

The following list of Synplify constraints are converted to the equivalent Quartus II SDC commands and are forward-annotated to the Quartus II software in the **.scf** file:

- `define_clock`

- define_input_delay
- define_output_delay
- define_multicycle_path
- define_false_path

All Synplify constraints described in the following sections use the same Synplify commands as described in "Passing Constraints to the Quartus II Software using Tcl Commands" on page 7–22; however, the constraints are mapped to SDC commands for the TimeQuest Timing Analyzer.



For the syntax and arguments for these commands, refer to the applicable subsection or refer to the Synplify Help. For a list of corresponding commands in the Quartus II software, refer to the Quartus II Help.

Individual Clocks and Frequencies

You can specify clock frequencies for individuals clocks in Synplify software with the command, `define_clock`. This command is passed to Quartus II software with `create_clock`.

Input and Output Delay

You can specify input delay and output delay constraints in Synplify software with the commands `define_input_delay` and `define_output_delay` respectively. These commands are passed to the Quartus II software with `set_input_delay` and `set_output_delay`.

Multicycle Path

You can specify a multicycle path constraint in Synplify with the command `define_multicycle_path`. This command is passed to the Quartus II software with `define_multicycle_path`.

False Path

You can specify a false path constraint in Synplify software with the command `define_false_path`. This command is passed to the Quartus II software with `set_false_path`.

Passing Constraints to the Quartus II Software using Tcl Commands

This section describes how Synplify constraints are converted to the equivalent Quartus II assignments and are forward-annotated to the Quartus II software with Tcl commands.

This section discusses timing constraints for the Quartus II Classic Timing Analyzer. If you are using the TimeQuest Timing Analyzer, the Quartus II timing constraints described in this section do not apply. Refer to "Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File" on page 7–20 for information about timing constraints supported by TimeQuest.

Global Signals

The Synplify software automatically promotes clock signals to global routing lines and passes **Global Signal** assignments to the Quartus II software. The assignments ensure that the same global routing constraints are applied during placement and routing.



The signals promoted to global routing can be different than the ones that the Quartus II software promotes to global routing by default. Synplify promotes only clock signals and not other control signals such as reset or enable. By default, without constraints from the Synplify software, the Quartus II software promotes control signals to global routing if they have high fan-out.

Default or Global Clock Frequency

Use the following Synplify command to set the Synplify default or global clock frequency that applies to the entire project:

```
set_option -frequency <frequency>
```

The <*frequency*> is specified in MHz. If a global frequency is not specified, the software uses the default global clock frequency of 1 MHz.

The `set_option` Synplify constraint is passed to the Quartus II software with the following command:

```
set_global_assignment -name FMAX_REQUIREMENT
<frequency> MHz
```

If a frequency is not specified in the Quartus II software, the software uses the default global clock frequency of 1 GHz.

Individual Clocks and Frequencies

You can specify clock frequencies for individual clocks with the following Synplify commands:

Example 7–6. Specifying Clock Frequencies for Individual Clocks

```
define_clock -name {<clock_name>} -freq <frequency> -clockgroup <clock_group>
-rise <rise_time> -fall <fall_time>
define_clock -name {<clock_name>} -period <period> -clockgroup <clock_group>
-rise <rise_time> -fall <fall_time>
```

Table 7–3 shows the command arguments.

Table 7–3. Command Arguments

Argument	Description
`-name`	The <*clock_name*> specifies a design port name or a register output signal name, and, after synthesis, corresponds to a <*mapped_clock_name*>.
`-freq` *(1)*	The <*frequency*> is specified in MHz.
`-period` *(2)*	The <*period*> is specified in ns.
`-clockgroup`	If the <*clock_group*> is not specified, it defaults to `default_clkgroup`. Synplify assumes all clocks belonging to the same clock group are related. If you do not specify a clock group, the clock belongs to the default clock group. Therefore, if you do not specify any clock groups, all the clocks are considered related by default in the software.
`-rise` `-fall`	The <*rise_time*> and <*fall_time*> specify a non-default duty cycle. By default, the Synplify synthesis tool assumes that the clock is a 50% duty cycle clock, with the rising edge at 0 and the falling edge at period/2. If you have another duty clock cycle, you can specify the appropriate **Rise At** and **Fall At** values.

Notes to Table 7–3:

(1) When the <*frequency*> is specified, the Synplify software uses <*fall_time*> and <*frequency*> to calculate the `duty_cycle` with the following formula: `duty_cycle` = (<*fall_time*> – <*rise_time*>) × <*frequency*> ÷ 10.

(2) When the <*period*> is specified, the Synplify software uses <*fall_time*> and <*period*> to calculate the `duty_cycle` with the following formula: `duty_cycle` = 100 × (<*fall_time*> – <*rise_time*>) ÷ <*period*>.

The equivalent Quartus II Classic Timing Analyzer commands depend on how the clock groups are defined. In the Quartus II software, clocks that belong to the same or related clock settings are considered related clocks. Clocks assigned to unrelated clock settings are unrelated clocks. There is a one-to-one correspondence between each Quartus II clock setting and a Synplify clock group.



The following sections describe only the frequency constraints. You can use the corresponding constraints for period.

Virtual Clocks

The Quartus II software supports virtual clocks. If you use the virtual clock setting in Synplify, the setting is mapped to a constraint in the Quartus II software.

Route Delay Option

The `-route` option in Synplify clock constraints is designed for use for synthesis only if you do not meet timing goals because the routing delay after placement and routing exceeds the delay predicted by the Synplify software. This constraint does not have to be forward annotated to the Quartus II software.

Multiple Clocks in Different Clock Groups

You can specify clock frequencies for multiple clocks with the Synplify commands shown in Example 7–7.

Example 7–7. Specifying Clock Frequencies for Multiple Clocks

```
define_clock -name {<clock_name1>} -freq <frequency1> \
-clockgroup <clock_group1> -rise <rise_time1> -fall <fall_time1>

define_clock -name {<clock_name2>} -freq <frequency2> \
-clockgroup <clock_group2> -rise <rise_time2> -fall <fall_time2>
```

<*clock_group1*> and <*clock_group2*> are unique names defined in the Synplify software for base clock settings in the Quartus II Classic Timing Analyzer.

If the clock <*rise_time*> is zero ("0"), multiple separate clocks are passed to the Quartus II software with the commands shown in Example 7–8:

Example 7–8. Quartus II Assignments for Multiple Clocks if the Clock Rise Time is Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty_cycle1> \
-target mapped_clock_name1 <base_clock_setting1>

create_base_clock -fmax <frequency2>MHz -duty_cycle <duty_cycle2> \
-target mapped_clock_name2 <base_clock_setting2>
```

If the clock <*rise_time*> is non-zero, multiple separate clocks are passed to the Quartus II software with the following commands shown in Example 7–9:

Example 7–9. Quartus II Assignments for Multiple Clocks if the Clock Rise Time is Not Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty cycle1> \
-no_target <base clock setting1>

create_base_clock -fmax <frequency2>MHz -duty_cycle <duty cycle2> \
-no_target <base clock setting2>

create_relative_clock -base_clock <base clock setting1> -offset <rise time1>ns \
-duty_cycle <duty cycle1> -multiply <multiply by> -divide <divide by> \
-target <mapped clock name1> <derived clock setting1>

create_relative_clock -base_clock <base clock setting2> -offset <rise time2>ns \
-duty_cycle <duty cycle2> -multiply <multiply by> -divide <divide by> \
-target <mapped clock name2> <derived clock_setting2>
```

Multiple Clocks with Different Frequencies in the Same Clock Group

You can specify multiple clocks with relative clock settings in the same clock group in Synplify with different frequencies with the commands shown in Example 7–10:

Example 7–10. Specifying Multiple Clocks with Different Frequencies in the Same Clock Group

```
define_clock -name {<clock_name1>} -freq <frequency1> -clockgroup <clock_group1> \
-rise <rise_time1> -fall <fall_time1>

define_clock -name {<clock_name2>} -freq <frequency2> -clockgroup <clock_group1> \
-rise <rise_time2> -fall <fall_time2>
```



When you specify clocks with different frequencies in the same clock group, the software calculates the <*multiply_by*> and the <*divide_by*> factors for relative clock settings from <*frequency1*> and <*frequency2*> in the clock group settings.

If the clock <*rise_time*> is zero, multiple clocks with relative clock settings in the same clock group with different frequencies are passed to the Quartus II software with the commands shown in Example 7–11:

Example 7–11. Quartus II Assignments for Multiple Clocks with Different Frequencies in the Same Clock Group, if the Clock Rise Time is Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty_cycle1> \
-target <mapped_clock_name1> <base_clock_setting1>

create_relative_clock -base_clock <base_clock_setting1> \
-duty_cycle <duty_cycle2> -multiply <multiply_by> -divide <divide_by> \
-target <mapped_clock_name2> <derived_clock_setting2>
```

Inter-Clock Relationships—Delays and False Paths between Clocks

You can set a clock-to-clock delay constraint in Synplify with the commands in Example 7–12.

Example 7–12. Specifying Clock-to-Clock Delay Constraints

```
define_clock_delay -fall <clock_name1> -rise <clock_name2> <delay_value>
define_clock_delay -rise <clock_name1> -fall <clock_name2> <delay_value>
define_clock_delay -rise <clock_name1> -rise <clock_name2> <delay_value>
define_clock_delay -fall <clock_name1> -fall <clock_name2> <delay_value>
```

If <*delay_value*> is set to `false`, these constraints in Synplify indicate a false path between the two clocks. If all four rise/fall clock-edge pairs are specified in the Synplify software, the Synplify constraints are mapped to the following constraint in the Quartus II software:

```
set_timing_cut_assignment -from <clock_name1> \
-to <clock_name2>
```

If all four clock-edge pairs are not specified in Synplify, the constraint cannot be mapped to a constraint for the Quartus II Classic Timing Analyzer.

If <*delay_value*> is set to a value other than `false`, these constraints in Synplify is not mapped to a constraint in the Quartus II software. The Quartus II Classic Timing Analyzer does not support clock-edge to clock-edge delay constraints.

False Paths

You can specify the false path constraint in Synplify with the following command:

```
define_false_path -from <sig_name1> -to <sig_name2>
```

The signals <*sig_name1*> and <*sig_name1*> can be design port names or register instance names.

The **define_false_path** constraint in Synplify is mapped to the constraint in the Quartus II software, as shown below.

```
set_timing_cut_assignment -from <sig_name1> \
-to <sig_name2>
```

Synplify can identify pairs of signal sets such that every member of the cross-product of these two sets is a valid false path constraint. Signal groups can be defined in the Quartus II Classic Timing Analyzer with the following commands:

```
timegroup -add_member sig_name1_i <sig_group1>
```
(for every signal in <*sig_group1*>)

```
timegroup -add_member sig_name2_i <sig_group2>
```
(for every signal in <*sig_group2*>)

```
set_timing_cut_assignment -from <sig_group1> \
-to <sig_group2>
```

If the signals <*sig_name1*> or <*sig_name2*> represent multiple signals such as a wildcard, group, or bus, the constraints you can expand appropriately for representation in the Quartus II software. The Quartus II software supports wildcard signal names, and signal groups for timing assignments. The Quartus II software does not support bus notation, such as **A[7:4]**.

False Path from a Signal
You can specify a false path constraint from a signal in Synplify with the following command:

```
define_false_path -from <sig_name>
```

The Quartus II Classic Timing Analyzer does not support "from-only" path specifications. You must also include a "to-path" specification. However, you can specify a wildcard for the `-to` signal. This constraint in Synplify is mapped to the following constraint in the Quartus II software:

```
set_timing_cut_assignment -from <sig_name> -to {*}
```

False Path to a Signal
You can specify a false path constraint to a signal in Synplify with the following command:

```
define_false_path -to <sig_name>
```

The Quartus II Classic Timing Analyzer does not support **to-only** path specifications. You must include a **from-path** specification." However, you can specify a wildcard for the `-from` signal. This constraint in Synplify is mapped to the following constraint in the Quartus II software:

```
set_timing_cut_assignment -from {*} -to <sig_name>
```

False Path Through a Signal
You can specify a false path constraint through a signal in Synplify with the following command:

```
define_false_path -from <sig_name1> -to <sig_name2> \
-through <sig_name3>
```

The Quartus II Classic Timing Analyzer does not support false paths with a "through path" specification. Any constraint in Synplify with a `-through` specification is not mapped to a constraint for the Quartus II Classic Timing Analyzer.

Multicycle Paths

You can specify a multicycle path constraint in Synplify with the following command:

```
define_multicycle_path -from <sig_name1> \
-to <sig_name2> <clock_cycles>
```

This constraint in Synplify is mapped to the following constraint in the Quartus II software:

```
set_multicycle_assignment -from <sig_name1> \
-to <sig_name2> <clock_cycles>
```

If the signals <*sig_name1*> or <*sig_name2*> represent multiple signals such as a wildcard, group, or bus, the constraints can be appropriately expanded for representation in the Quartus II software as described in "False Paths" on page 7–11.



<*clock_cycles*> is the number of clock cycles for the multicycle path.

Multicycle Path from a Signal

You can specify a multicycle path constraint from a signal in Synplify with the following command:

```
define_multicycle_path -from <sig_name> <clock_cycles>
```

This constraint is mapped using a wildcard for the `-to` value in the Quartus II Classic Timing Analyzer, similar to the false path constraints:

```
set_multicycle_assignment -from <sig_name> \
-to {*} <clock_cycles>
```

Multicycle Path to a Signal

You can specify a multicycle path constraint to a signal in Synplify with the following command:

```
define_multicycle_path -to <sig_name> <clock_cycles>
```

This constraint is mapped using a wildcard for the `-from` value in the Quartus II Classic Timing Analyzer, similar to the false path constraints:

```
set_multicycle_assignment -from {*} <sig_name> \
<clock_cycles>
```

Multicycle Path Through a Signal

You can specify a multicycle path constraint through a signal in Synplify using the following command:

```
define_multicycle_path -from <sig_name1> -to <sig_name2> \
-through <sig_name3> <clock_cycles>
```

The Quartus II Classic Timing Analyzer does not support multicycle paths with a "through path" specification. Any constraint in Synplify with a `-through` specification is not mapped to a constraint for the Quartus II Classic Timing Analyzer.

Maximum Path Delays

You can specify the maximum path delay relationships between signals in Synplify with the following command:

```
define_path_delay -from <sig_name1> -to <sig_name2> \
-max <delay_value>
```

This constraint in Synplify is mapped to the following constraint in the Quartus II software:

```
set_instance_assignment -from <sig_name1> \
-to <sig_name2> -name SETUP_RELATIONSHIP <delay_value>ns
```

The Quartus II Classic Timing Analyzer does not support signal groups or bus notation, and supports only register names for this constraint.

Maximum Path Delay from a Signal
You can specify the maximum path delay constraint from a signal in Synplify with the following command:

```
define_path_delay -from <sig_name> -max <delay_value>
```

This constraint is mapped using a wildcard for the `-to` value in the Quartus II Classic Timing Analyzer, similar to false path constraints:

```
set_instance_assignment -from <sig_name> -to {*} \
-name SETUP_RELATIONSHIP <delay_value>ns
```

Maximum Path Delay to a Signal
You can specify the maximum path delay constraint to a signal in Synplify with the following command:

```
define_path_delay -to <sig_name> -max <delay_value>
```

This constraint is mapped using a wildcard for the `-from` value in the Quartus II Classic Timing Analyzer, similar to the false path constraints.

```
set_instance_assignment -from {*}<sig_name> \
-to <sig_name> -name SETUP_RELATIONSHIP <delay_value>ns
```

Maximum Path Delay through a Signal
You can specify the maximum path delay constraint through a signal in Synplify with the following command:

```
define_path_delay -from <sig_name1> -to <sig_name2> \
-through <sig_name3> -max <delay_value>
```

The Quartus II Classic Timing Analyzer does not support maximum path delay constraints with a "through path" specification. Any constraint in Synplify with a `-through` specification is not mapped to a constraint for the Quartus II Classic Timing Analyzer.

Register Input and Output Delays
These register input delay and register output delay constraints in Synplify are for use in synthesis only, and therefore are not forward-annotated to the Quartus II software.

Default External Input Delay
You can specify the default input delay constraint in Synplify with the following command:

```
define_input_delay -default <delay_value>
```

This constraint is mapped to the following constraint in the Quartus II software:

```
set_input_delay -clock {*} <delay_value> {*}
```

Port-Specific External Input Delay
You can specify a port-specific input delay constraint in Synplify with the following command:

```
define_input_delay <input_port_name> <delay_value> \
-ref <clock_name>:<clock_edge>
```

The <*clock_edge*> can be set to `r` (rising edge) or `f` (falling edge).

When the clock edge is `r` (rising edge), this constraint is mapped to the following constraint in the Quartus II software:

```
set_input_delay -clock <clock_name> <delay_value> \
<input_port_name>
```

When the clock_edge is `f` (falling edge), this constraint is not mapped to a constraint in the Quartus II software. The Quartus II Classic Timing Analyzer does not support the specification of input delays with respect to the falling edge of the clock.

Default External Output Delay
You can specify the default output delay constraint in Synplify with the following command:

```
define_output_delay -default <delay_value>
```

This constraint is mapped to the following constraint in the Quartus II software:

```
set_output_delay -clock {*} <delay_value> {*}
```

Port-Specific External Output Delay
You can specify a port-specific input delay constraint in Synplify with the following command:

```
define_output_delay <output_port_name> <delay_value> \
-ref <clock_name>:<clock_edge>
```

The <*clock_edge*> can be set to `r` (rising edge) or `f` (falling edge). When the clock edge is `r` (rising edge), this constraint is mapped to the following constraint in the Quartus II software:

```
set_output_delay -clock <clock_name> <delay_value> \
<output_port_name>
```

When the clock_edge is `f` (falling edge), this constraint is not mapped to a constraint in the Quartus II software. The Quartus II Classic Timing Analyzer does not support the specification of output delays with respect to the falling edge of the clock.

Guidelines for Altera Megafunctions and Architecture-Specific Features

Altera provides parameterizable megafunctions including the LPMs, device-specific Altera megafunctions, IP available as Altera MegaCore® functions, and IP available through the Altera Megafunction Partners Program (AMPP℠). You can use megafunctions by instantiating them in your HDL code or inferring them from generic HDL code.

If you want to instantiate a megafunction in your HDL code, you can do so with the MegaWizard Plug-In Manager to parameterize the function or instantiating the function using the port and parameter definition. The MegaWizard Plug-In Manager provides a graphical interface within the Quartus II software for customizing and parameterizing any available megafunction for the design. "Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager" on page 7–33 describes the MegaWizard Plug-In Manager flow with the Synplify software.

For more information about specific Altera megafunctions, refer to the Quartus II Help. For more information about IP functions, refer to the appropriate IP documentation.

The Synplify software also automatically recognizes certain types of HDL code and infers the appropriate megafunction when a megafunction provides optimal results. The Synplify software provides options to control inference of certain types of megafunctions, as described in "Inferring Altera Megafunctions from HDL Code" on page 7–37.

For a detailed discussion about instantiating versus inferring megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. The *Recommended HDL Coding Styles* chapter also provides details on using the MegaWizard Plug-In Manager in the Quartus II software and explains the files generated by the wizard, as well as providing coding style recommendations and HDL examples for inferring megafunctions in Altera devices.

Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager

When you use the MegaWizard Plug-In Manager to set up and parameterize a megafunction, the MegaWizard Plug-In Manager creates a VHDL or Verilog HDL wrapper file that instantiates the megafunction.

Beginning with the Quartus II software version 7.1, there is an option in the MegaWizard Plug-In Manager to create a netlist for area and timing estimation instead of a wrapper file. This option is not supported with the Synplify software version 8.8, therefore you must use the megafunction wrapper file as described in this section.

Using the MegaWizard Plug-In Manager-generated wrapper file is referred to as a black box methodology because the megafunction is treated as a black box in the Synplify software. The black box methodology does not allow the synthesis tool any visibility into the function module and therefore does not take full advantage of the synthesis tool's timing-driven optimization. For better timing optimization, especially if the black box does not have registered inputs and outputs, add timing models to black boxes. Refer to "Other Synplify Software Attributes for Creating Black Boxes" on page 7–36 for details.

Make sure to set the correct Quartus II version in the Synplify software before compiling the Megawizard-generated file. The Quartus version setting should match the version of the Quartus II software used to generate the customized megafunction in the MegaWizard Plug-In Manager.

1. On the Project menu, click **Implementation Options**.
2. Click the Implementation Results tab, then click **Quartus Version**.
3. Choose the correct version number in the list.

Alternately, use the following command from the command line:

```
set_option -quartus_version <version number> ↵
```

Using MegaWizard Plug-In Manager-Generated Verilog HDL Files for Black Box Megafunction Instantiation

If you check the <*output file*>**_inst.v** and <*output file*>**_bb.v** options on the last page of the wizard, the MegaWizard Plug-In Manager generates a Verilog HDL instantiation template file and a hollow-body black box module declaration for use in your Synplify design. The instantiation template file helps to instantiate the megafunction variation wrapper file, <*output file*>**.v**, in your top-level design. Do not include the megafunction variation wrapper file in your Synplify Project, but add it, with your Synplify-generated VQM netlist, to your Quartus II project. Add the hollow-body black box module declaration <*output file*>**_bb.v** to your Synplify project to describe the port connections of the black box.



The Synplify software reads black box instantiations for the `alt_pll` megafunction and writes the phase-locked loop (PLL) instance into the resulting VQM output netlist. Reading the PLL function allows the Synplify software to interpret the multiplication and division factors in the PLL instantiation to make the correct timing assignments. Therefore, for `alt_pll` instantiations, make sure to include the megafunction variation wrapper file <*output file*>**.v** in your Synplify project and do not declare it as a black box. Because the instance is written in the VQM file, do not include the `alt_pll` megafunction variation wrapper file <*output file*>**.v** in your Quartus II project.

You can use the `syn_black_box` compiler directive to declare a module as a black box. The top-level design files must contain the megafunction port mapping and hollow-body module declaration, as described above. You can apply the `syn_black_box` directive to the module declaration in the top-level file or a separate file included in the project (such as the <*output file*>**_bb.v** file) to instruct the Synplify software that this is a black box. The software compiles successfully without this directive, but reports an additional warning message. Using this directive allows you to add other directives as discussed in "Other Synplify Software Attributes for Creating Black Boxes" on page 7–36.

Example 7–13 shows a sample top-level file that instantiates **verilogCount.v**, which is a customized variation of the `lpm_counter` generated by the MegaWizard Plug-In Manager.

Example 7–13. Top-Level Verilog HDL Code with Black Box Instantiation of lpm_counter

```
module topCounter (clk, count);
    input clk;
    output[7:0]  count;
    verilogCounter verilogCounter_inst (
        .clock ( clk ),
        .q ( count )
);
endmodule
// Module declaration found in verilogCounter_bb.v
// The following attribute is added to create a
// black box for this module.
module verilogCounter (
    clock,
    q) /* synthesis syn_black_box */;
    input clock;
    output[7:0]  q;
endmodule
```

Using MegaWizard Plug-In Manager-Generated VHDL Files for Black Box Megafunction Instantiation

If you check the <*output file*>**.cmp** and <*output file*>**_inst.vhd** options on the last page of the wizard, the MegaWizard Plug-In Manager generates a VHDL component declaration file and a VHDL instantiation template file for use in your Synplify design. These files can help you instantiate the megafunction variation wrapper file, <*output file*>**.vhd**, in your top-level design. Do not include the megafunction variation wrapper file in your Synplify project, but add it, along with your Synplify-generated VQM netlist, to your Quartus II project.



The Synplify software reads black box instantiations for the `alt_pll` megafunction and writes the phase-locked loop (PLL) instance into the resulting VQM output netlist. Reading the PLL function allows the Synplify software to interpret the multiplication and division factors in the PLL instantiation to make the correct timing assignments. Therefore, for `alt_pll` instantiations, make sure to include the megafunction variation wrapper file <*output file*>**.vhd** in your Synplify project and do not declare it as a black box. Because the instance is written in the VQM file, do not include the `alt_pll` megafunction variation wrapper file <*output file*>**.vhd** in your Quartus II project.

You can use the syn_black_box compiler directive to declare a component as a black box. The top-level design files must contain the megafunction variation component declaration and port mapping, as described above. Apply the syn_black_box directive to the component declaration in the top-level file. The software compiles successfully without this directive, but reports an additional warning message. Using this directive allows you to add other directives such as the ones in the "Other Synplify Software Attributes for Creating Black Boxes" section.

Example 7–14 shows a sample top-level file that instantiates **vhdlCount.vhd**, which is a customized variation of the lpm_counter generated by the MegaWizard Plug-In Manager.

Example 7–14. Top-Level VHDL Code with Black Box Instantiation of lpm_counter

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY testCounter IS
    PORT
    (
        clk: IN STD_LOGIC ;
        count: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
    END testCounter;
ARCHITECTURE top OF testCounter IS
component vhdlCount
    PORT (
        clock: IN STD_LOGIC ;
        q: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
end component;
attribute syn_black_box : boolean;
attribute syn_black_box of vhdlCount: component is true;
BEGIN
    vhdlCount_inst : vhdlCount PORT MAP (
        clock => clk,
        q => count
    );
END top;
```

Other Synplify Software Attributes for Creating Black Boxes

The black box methodology does not allow the synthesis tool any visibility into the function module. Thus, it does not take full advantage of the synthesis tool's timing-driven optimization. For better timing optimization, especially if the black box does not have registered inputs and outputs, add timing models to black boxes. This can be done by adding the syn_tpd, syn_tsu, and syn_tco attributes. Refer to Example 7–15 for a Verilog HDL example.

Example 7–15. Verilog HDL Example

```
module ram32x4(z,d,addr,we,clk);
/* synthesis syn_black_box syn_tco1="clk->z[3:0]=4.0"
    syn_tpd1="addr[3:0]->z[3:0]=8.0"
    syn_tsu1="addr[3:0]->clk=2.0"
    syn_tsu2="we->clk=3.0" */
output[3:0]z;
input[3:0]d;
input[3:0]addr;
input we
input clk
endmodule
```

The following additional attributes are supported by the Synplify software to communicate details about the characteristics of the black box module within the HDL code:

- `syn_resources`—Specifies the resources used in a particular black box
- `black_box_pad_pin`—Prevents mapping to I/O cells
- `black_box_tri_pin`—Indicates a tri-stated signal



For more information about applying these attributes, refer to the *Tasks and Tips* chapter of the *Synplify User Guide*.

Inferring Altera Megafunctions from HDL Code

The Synplify software uses Behavior Extraction Synthesis Technology (BEST) algorithms to infer high-level structures such as RAMs, ROMs, operators, FSMs, and so forth. It then keeps the structures abstract for as long as possible in the synthesis process. This allows the use of technology-specific resources to implement these structures by inferring the appropriate Altera megafunction when a megafunction provides optimal results. The following sections outline some of the Synplify-specific details when inferring Altera megafunctions. The Synplify software provides options to control inference of certain types of megafunctions, which is also described in the following sections.

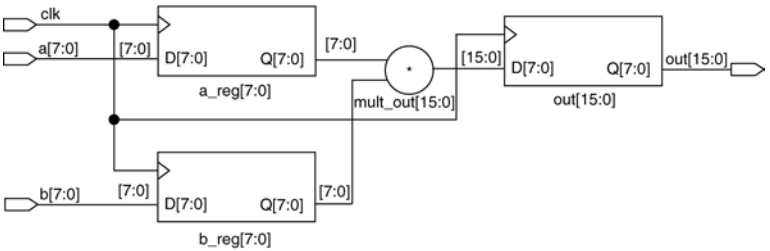


For coding style recommendations and examples for inferring megafunctions in Altera devices, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Inferring Multipliers

Figure 7–2 shows the HDL Analyst view of an unsigned 8 × 8 multiplier with two pipeline stages after synthesis as seen in HDL Analyst in the Synplify software. This multiplier is converted into an `lpm_mult` megafunction. For devices with DSP blocks, the software may implement the `lpm_mult` function in a DSP block instead of regular logic, depending on device utilization. For Stratix II and Stratix III devices, the software maps directly to DSP block device atoms instead of instantiating a megafunction in the **.vqm** file.

Figure 7–2. HDL Analyst View of lpm_mult Megafunction (Unsigned 8 × 8 Multiplier with Pipeline=2)



Resource Balancing

While mapping multipliers to DSP blocks, the Synplify software performs resource balancing for optimum performance.

Altera devices have a fixed number of DSP blocks, which include a fixed number of embedded multipliers. If the design uses more multipliers than are available, the Synplify software automatically maps the extra multipliers to logic elements (LEs), or adaptive logic modules (ALMs).

If a design uses more multipliers than are available in the DSP blocks, the Synplify software maps the multipliers in the critical paths to DSP blocks. Next, any wide multipliers, which may or may not be in the critical paths, are mapped to DSP blocks. Smaller multipliers and multipliers that are not in the critical paths may then be implemented in the logic (LEs or ALMs). This ensures that the design fits successfully in the device.

Controlling the Inferring of DSP Blocks

Multipliers can be implemented in DSP blocks or in logic in Altera devices that contain DSP blocks. You can control this implementation through attribute settings in the Synplify software.

Signal Level Attribute

You can control the implementation of individual multipliers by using the `syn_multstyle` attribute as shown below:

```
<signal_name> /* synthesis syn_multstyle = "logic" */
```

where *signal_name* is the name of the signal.



This setting applies to wires only; it cannot be applied to registers.

Table 7–4 shows the values for the signal level attribute in the Synplify software that controls the implementation of the multipliers in the DSP blocks or LEs.

Table 7–4. Attribute Settings for DSP Blocks in the Synplify Software

Attribute Name	Value	Description
`syn_multstyle`	`lpm_mult`	LPM Function inferred and multipliers implemented in DSP blocks
`syn_multstyle`	`logic`	LPM function not inferred and multipliers implemented LEs by the Synplify software
`syn_multstyle`	`block_mult`	DSP megafunction is inferred and multipliers are mapped directly to DSP block device atoms (for supported devices)

Example 7–16 and Example 7–17 show simple Verilog HDL and VHDL code using the `syn_multstyle` attribute.

Example 7–16. Signal Attributes for Controlling DSP Block Inference in Verilog HDL Code

```
module mult(a,b,c,r,en);

input [7:0] a,b;
output [15:0] r;
input [15:0] c;
input en;
wire [15:0] temp /* synthesis syn_multstyle="logic" */;

assign temp = a*b;
assign r = en ? temp : c;
endmodule
```

Example 7–17. Signal Attributes for Controlling DSP Block Inference in VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity onereg is port (
    r  : out std_logic_vector(15 downto 0);
    en : in std_logic;
    a  : in std_logic_vector(7 downto 0);
    b  : in std_logic_vector(7 downto 0);
    c  : in std_logic_vector(15 downto 0)
    );
end onereg;

architecture beh of onereg is
signal temp : std_logic_vector(15 downto 0);
attribute syn_multstyle : string;
attribute syn_multstyle of temp : signal is "logic";

begin
temp <= a * b;
    r <= temp when en='1' else c;
end beh;
```

Inferring RAM

When a RAM block is inferred from an HDL design, the software uses an Altera megafunction to target the device memory architecture. For Stratix II and Stratix III devices, the software maps directly to memory block device atoms instead of instantiating a megafunction in the VQM file.

Follow the guidelines below for the Synplify software to successfully infer RAM in a design:

- The address line must be at least two bits wide.
- Resets on the memory are not supported. Refer to the device family documentation for information about whether read and write ports must be synchronous.
- Some Verilog HDL statements with blocking assignments may not be mapped to RAM blocks, so avoid blocking statements when modeling RAMs in Verilog HDL.

For certain device families, the `syn_ramstyle` attribute specifies the implementation to use for an inferred RAM. You can apply `syn_ramstyle` globally, to a module, or to a RAM instance, to specify `registers` or `block_ram` values. To turn off RAM inference, set the attribute value to `registers`.

When inferring RAM for certain Altera device families, the Synplify software generates additional bypass logic. This logic is generated to resolve a half-cycle read/write behavior difference between the RTL and post-synthesis simulations. The RTL simulation shows the memory being updated on the positive edge of the clock, and the post-synthesis simulation shows the memory being updated on the negative edge. To eliminate the bypass logic, the output of the RAM must be registered. By adding this register, the output of the RAM is seen after a full clock cycle, by which time the update has occurred; thus, eliminating the need for the bypass logic.

For devices with TriMatrix memory blocks, you can disable the creation of glue logic by setting the `syn_ramstyle` value to `no_rw_check`. Use `syn_ramstyle` with a value of `no_rw_check` to disable the creation of glue logic in dual-port mode.

Example 7–18 shows sample VHDL code for inferring dual-port RAM.

Example 7–18. VHDL Code for Inferred Dual-Port RAM

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY dualport_ram IS
PORT ( data_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    data_in: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    wr_addr, rd_addr: IN STD_LOGIC_VECTOR (6 DOWNTO 0);
    we: IN STD_LOGIC;
    clk: IN STD_LOGIC);
END dualport_ram;

ARCHITECTURE ram_infer OF dualport_ram IS
TYPE Mem_Type IS ARRAY (127 DOWNTO 0) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL mem: Mem_Type;
SIGNAL addr_reg: STD_LOGIC_VECTOR (6 DOWNTO 0);

BEGIN
    data_out <= mem (CONV_INTEGER(rd_addr));
    PROCESS (clk, we, data_in) BEGIN
        IF (clk='1' AND clk'EVENT) THEN
            IF (we='1') THEN
            mem(CONV_INTEGER(wr_addr)) <= data_in;
            END IF;
        END IF;
    END PROCESS;
END ram_infer;
```

Example 7–19 shows an example of the VHDL code preventing bypass logic for inferring dual-port RAM. The extra latency behavior stems from the inferring methodology and is not required when instantiating a megafunction.

Example 7–19. VHDL Code for Inferred Dual-Port RAM Preventing Bypass Logic

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY dualport_ram IS
PORT ( data_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    data_in : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    wr_addr, rd_addr : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
    we : IN STD_LOGIC;
    clk : IN STD_LOGIC);
END dualport_ram;

ARCHITECTURE ram_infer OF dualport_ram IS
TYPE Mem_Type IS ARRAY (127 DOWNTO 0) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL mem : Mem_Type;
SIGNAL addr_reg : STD_LOGIC_VECTOR (6 DOWNTO 0);
SIGNAL tmp_out : STD_LOGIC_VECTOR(7 DOWNTO 0); --output register

BEGIN
    tmp_out <= mem (CONV_INTEGER(rd_addr));
    PROCESS (clk, we, data_in) BEGIN
        IF (clk='1' AND clk'EVENT) THEN
            IF (we='1') THEN
            mem(CONV_INTEGER(wr_addr)) <= data_in;
            END IF;
            data_out <= tmp_out; --registers output preventing
                                 -- bypass logic generation.
        END IF;
    END PROCESS;
END ram_infer;
```

RAM Initialization

You can use Verilog system tasks `$readmemb` or `$readmemh` in your HDL code to initialize RAM memories. The Synplify compiler forward-annotates the initialization values in the **.srs** (technology-independent RTL netlist) file and the mapper generates a corresponding hexadecimal memory initialization (**.hex**) file. One HEX file is created for each of the `altsyncram` megafunctions that are inferred in the design. The HEX file is associated with the `altsyncram` instance in the **.vqm** file using the `init_file` attribute.

Example 7–20 and Example 7–21 illustrate how RAM memories can be initialized through HDL code, and how the corresponding HEX file is generated using Verilog HDL.

Example 7–20. Using $readmemb System Task to Initialize an Inferred RAM in Verilog HDL Code

```
initial
begin
      $readmemb("mem.ini", mem);
end

always @(posedge clk)
begin
    raddr_reg <= raddr;
    if(we)
        begin
            mem[waddr] <= data;
        end
end
```

Example 7–21. Sample VQM Instance Containing Memory Initialization File from Example 7–20

```
altsyncram mem_hex( .wren_a(we), .wren_b(GND),...);

defparam mem_hex.lpm_type = "altsyncram";
defparam mem_hex.operation_mode = "Dual_Port";
...
defparam mem_hex.init_file = "mem_hex.hex";
```

Inferring ROM

When a ROM block is inferred from an HDL design, the software uses an Altera megafunction to target the device memory architecture. For Stratix II and Stratix III devices, the software maps directly to memory block device atoms instead of instantiating a megafunction in the VQM file. Follow the guidelines below for the Synplify software to successfully infer ROM in a design:

- The address line must be at least two bits wide.
- ROM must be at least half full.
- A CASE or IF statement must make 16 or more assignments using constant values of the same width.

Inferring Shift Registers

The software infers shift registers for sequential shift components so that they can be placed in dedicated memory blocks in supported device architectures using the `altshift_taps` megafunction.

If required, set the implementation style with the `syn_srlstyle` attribute. If you do not want the components automatically mapped to shift registers, set the value to `registers`. You can set the value globally or on individual modules or registers.

For some designs, turning off shift register inference can improve the design performance.

Incremental Compilation and Block-Based Design

As designs become more complex and designers work in teams, a block-based hierarchical or incremental design flow is often an effective design approach. In an incremental compilation flow, you can make changes to part of the design while maintaining the placement and performance of unchanged parts of the design. Design iterations are made dramatically faster by focusing new compilations on a particular design partitions and merging results with previous compilation results of other partitions. In a bottom-up or team-based approach, you can perform optimization on individual subblocks and then preserve the results before you integrate the blocks into a final design and optimize it at the top level.

MultiPoint synthesis, which is available for certain device technologies in the Synplify Pro software, provides an automated block-based incremental synthesis flow. The MultiPoint feature manages a design hierarchy to let you design incrementally and synthesize designs that take too long for top-down synthesis of the entire project. MultiPoint synthesis allows different netlist files to be created for different sections of a design hierarchy, and supports Quartus II incremental compilation and LogicLock design methodologies. It also ensures that only those sections of a design that have been updated are resynthesized when the design is compiled, reducing synthesis run time and preserving the results for the unchanged blocks. You can change and resynthesize one section of a design without affecting other sections of the design.

You can also partition your design and create different netlist files manually with the Synplify software (basic Synplify and Synplify Pro) by creating a separate project for the logic in each partition of the design. Creating different netlist files for each partition of the design means that each partition is independent of the others. When synthesizing the entire project, only portions of a design that have been updated have to be resynthesized when you compile the design. You can make changes and resynthesize one partition of a design to create a new netlist file without affecting the synthesis results and placement of other partitions.

Hierarchical design methodologies can improve the efficiency of your design process, providing better design reuse opportunities and fewer integration problems when working in a team environment. When you use these incremental synthesis methodologies, you can take advantage

of the incremental compilation and methodologies in the Quartus II software. You can perform placement and routing on only the changed partitions of the design, reducing place-and-route time and preserving your fitting results. Following the guidelines in this section can help you achieve good results with these methodologies.

The following list shows the general top-down compilation flow when using these features of the Quartus II software:

1. Create Verilog HDL or VHDL design files as in the regular design flow.
2. Determine which hierarchical blocks are to be treated as separate partitions in your design.
3. Set up your design using the MultiPoint feature or separate projects so that a separate netlist file is created for each partition of the design.
4. If using separate projects, disable I/O pad insertion in the implementations for lower-level partitions.
5. Compile and technology-map each partition in the Synplify Pro or Synplify software, making constraints as you would in the regular design flow.
6. Import the VQM or EDIF netlist and the Tcl file for each partition into the Quartus II software and set up the Quartus II project(s) to use incremental compilation.
7. Compile your design in the Quartus II software and preserve the compilation results using the post-fit netlist in incremental compilation.
8. When you make design or synthesis optimization changes to part of your design, resynthesize only the changed partition to generate a new netlist and Tcl file. Do not regenerate netlist files for the unchanged partitions.
9. Import the new netlist and Tcl file into the Quartus II software and recompile the design in the Quartus II software using incremental compilation.



For more information about creating partitions and using the incremental compilation in the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Hierarchy and Design Considerations with Multiple VQM Files

To ensure the proper functioning of the synthesis flow, you can create separate netlist files only for modules and entities. In addition, each module or entity should have its own design file. If two different modules are in the same design file but are defined as being part of different partitions, you cannot maintain incremental compilation since both partitions would have to be recompiled when you change one of the modules.

Altera recommends that you register all inputs and outputs of each partition. This makes logic synchronous and avoids any delay penalty on signals that cross partition boundaries.

If you use boundary tri-states in a lower-level block, the Synplify software pushes (or "bubbles") the tri-states through the hierarchy to the top level to make use of the tri-state drivers on output pins of Altera devices. Because bubbling tri-states requires optimizing through hierarchies, lower-level tri-states are not supported with a block-based compilation methodology. You should use tri-state drivers only at the external output pins of the device and in the top-level block in the hierarchy.

Creating a Design with Separate Netlist Files

The first stage of a hierarchical or incremental design flow is to ensure that different parts of your design do not affect each other. Ensure that you have separate netlists for each partition in your design so that you can take advantage of incremental compilation in the Quartus II software. If the entire design is in one netlist file, changes in one partition may affect other partitions because of possible node name changes when you resynthesize the design.

You can generate multiple VQM files either by using the MultiPoint synthesis flow and LogicLock attributes in the Synplify Pro software, or by manually creating separate Synplify projects and creating a black box for each block that you want to be considered as a separate design partition.

In the MultiPoint synthesis flow (Synplify Pro only), you create multiple VQMs from one easy-to-manage, top-level synthesis project. By using the manual black box method (Synplify or Synplify Pro), you have multiple synthesis projects, which may be required for certain team-based or bottom-up designs where a single top-level project is not desired.

Once you have created multiple VQM files using one of these two methods, you must create the appropriate Quartus II projects to place-and-route the design.

Creating a Design with Multiple VQM Files Using Synplify Pro MultiPoint Synthesis

This section describes how to generate multiple VQM files using the Synplify Pro MultiPoint synthesis flow. You must first set up your compile points, constraint files, and Synplify Pro options, then apply Altera-specific attributes to write multiple VQM files and create LogicLock region assignments.

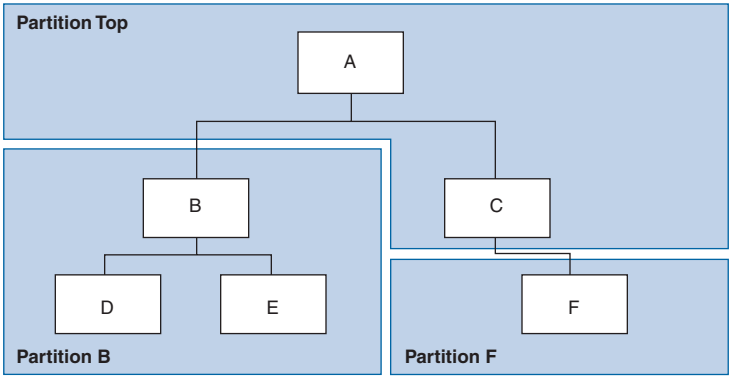
Set Compile Points and Create Constraint Files

The MultiPoint flow lets you segment a design into smaller synthesis units, called "Compile Points." The synthesis software treats each Compile Point as a partition for incremental mapping, which allows you to isolate and work on individual Compile Point modules as independent segments of the larger design without impacting other design modules. A design can have any number of Compile Points, and Compile Points can be nested. The top-level module is always treated as a Compile Point.

Compile Points are optimized in isolation from their parent, which can be another Compile Point or a top-level design. Each block created with a Compile Point is unaffected by critical paths or constraints on its parent or other blocks. A Compile Point is independent, with its own individual constraints. During synthesis, any Compile Points that have not yet been synthesized are synthesized before the top level. Nested Compile Points are synthesized before the parent Compile Points in which they are contained. When you apply the appropriate Synplify Pro LogicLock constraints to a Compile Point module, then a separate netlist is created for that Compile Point, isolating that logic from any other logic in the design.

Figure 7–3 on page 7–48 shows an example of a design hierarchy that can be split into multiple partitions. The top-level block of each partition can be synthesized as a separate Compile Point.

Figure 7–3. Partitions in a Hierarchical Design



In this case, modules A, B, and F are Compile Points. The top-level Compile Point consists of the top-level block in the design (that is, block A in this example), including the logic that is not defined under another Compile Point. In this example, the design for top-level Compile Point A also includes the logic in one of its subblocks, C. Because block F is defined as its own Compile Point, it is not treated as part of the top-level Compile Point A. Another separate Compile Point B contains the logic in blocks B, D, and E. One netlist is created for the top-level module A and submodule C, another netlist is created for B and its submodules D and E, while a third netlist is created for F.

Apply Compile Points to the module or architecture in the Synplify Pro SCOPE spreadsheet or in the **.sdc** file. You cannot set a Compile Point in the Verilog HDL or VHDL source code. You can set the constraints manually using Tcl or by editing the **.sdc** file. You can also use the GUI which provides two methods, manual or automated, as shown below.

Defining Compile Points Using Tcl or SDC

To set Compile Points using Tcl or an **.sdc** file, use the `define_compile_point` command, as shown in Example 7–22.

Example 7–22. The define_compile_point Command

```
define_compile_point [-disable] [-comment <comment>] <objname> \
[-type <compile point type>]
```

In the syntax statement above, *objname* represents any module in the design. Currently, `locked` is the only Compile Point type supported.

Each Compile Point has a set of constraint files that begin with the `define_current_design` command to set up the SCOPE environment, as shown below.

```
define_current_design {<my_module>}
```

Manually Defining Compile Points from the GUI

The manual method requires you to separately create constraint files for the top-level and the lower-level Compile Points. To use the manual method:

1. From the top level, select the **Compile Points** tab in the SCOPE spreadsheet.
2. Select the modules that you want to define as Compile Points.

   Currently, locked Compile Points are the only type supported. All Compile Points must be defined from the top level because the **Compile Points** tab is not available in the SCOPE spreadsheet from lower level modules.
3. Manually create a constraint file for each module.

To ensure that changes to a Compile Point do not affect the top-level parent module, disable the **Update Compile Point Timing Data** option on the **Implementation Options** dialog box. If this option is enabled, updates to a child module can impact the top-level module.

Automatically Defining Compile Points from the GUI

When you use the automated process, the lower-level constraint file is created automatically. This eliminates the manual step necessary to set up each Compile Point. To use the automated method, perform the following steps:

1. On the File menu, select **New**. Click to create a new **Constraint File**, or click the **SCOPE** icon in the tool bar.
2. From the **Select File Type** tab of the **Create a New SCOPE File** dialog box, select **Compile Point**.
3. Select the module you want to designate as a Compile Point. The software automatically sets the Compile Points in the top-level constraint file and creates a lower-level constraint file for each Compile Point.

To ensure that changes to a Compile Point do not affect the top-level parent module, disable the **Update Compile Point Timing Data** option on the **Implementation Options** dialog box. If this option is enabled, updates to a child module can impact the top-level module.

When using Compile Points with incremental compilation, keep the following restrictions in mind:

- To use Compile Points effectively, you must provide timing constraints (timing budgeting) for each Compile Point; the more accurate the constraints, the better your results are. Constraints are not automatically budgeted, so manual time budgeting is essential. Altera recommends that you register all inputs and outputs of each partition. This avoids any logic delay penalty on signals that cross partition boundaries.
- When using the Synplify Pro attribute `syn_useioff` to pack registers in the I/O Elements (IOEs) of Altera devices, these registers must be in the top-level module, not a lower level. Otherwise, you must allow the Quartus II software to perform I/O register packing instead of the `syn_useioff` attribute. You can use the **Fast Input Register** or **Fast Output Register** options, or set I/O timing constraints and turn on **Optimize I/O cell register placement for timing** on the Fitter Settings page of the **Settings** dialog box in the Quartus II software.
- There is no incremental synthesis support for top-level logic; any logic in the top-level is resynthesized during every compilation in the Synplify Pro software.



For more information about Compile Points, refer to the *Synplify Pro User Guide and Reference Manual* on the Synplicity web site at **www.synplicity.com**.

Apply the LogicLock Attributes

To instruct the Synplify Pro software to create a separate VQM netlist file for each Compile Point, you must indicate that the Compile Point is being used with LogicLock regions in the incremental compilation or LogicLock design methodology. Because separate netlist files are required for incremental compilation, you must use the LogicLock attributes if you plan to use the incremental compilation feature in the Quartus II software. When you apply the appropriate LogicLock attributes, the Synplify Pro software also writes out Tcl commands for the Quartus II software to create a LogicLock region for each netlist.

LogicLock regions in the Quartus II software have size and location properties. The region's size is defined by the height and width of the rectangular area. If the region is specified as auto-size, then the Quartus II

software determines the appropriate size to fit the logic assigned to the region. When you specify the size, you must include enough device resources to accommodate the assigned logic. The location of a region is defined by its origin, which is the position of its bottom-left corner or top-left corner, depending on the target device family. In the Quartus II software, this location can be specified as "locked" or "floating." If the location is floating, the Quartus II software determines the location during its optimization process.



Floating locations are the only type currently supported in the Synplify Pro software.

When you use incremental compilation in the Quartus II software, you should lock down the size and location of the region in the Quartus II software after the first compilation to achieve the best quality of results.

Table 7–5 shows the valid combinations of the LogicLock attributes.

Table 7–5. LogicLock Location and Size Properties

altera_logiclock_location Attribute	altera_logiclock_size Attribute	Description
Floating	Auto	The most flexible type of LogicLock constraint. Allows the Quartus II software to choose the appropriate region size and location.
Floating	Fixed	Assumes the size of LogicLock constraint area is already optimal in the existing Quartus II project.

You can apply these attributes to the top-level constraint file or to the individual constraint files for each lower-level Compile Point. You can use the **Attribute** tab of the SCOPE spreadsheet to set attributes.

Synplify Pro offers another attribute, `syn_allowed_resources`, which restricts the number of resources for a given module. You can apply the `syn_allowed_resources` attribute to any Compile Point view.



For specific information about these attributes, refer to the Synplify Pro online help or reference manual.

Creating a Quartus II Project for Multiple VQM Files

During compilation, the Synplify Pro software creates a <*top-level project*>**.tcl** file that provides the Quartus II software with the appropriate constraints and LogicLock assignments, creating a region for each VQM file along with the information to set up a Quartus II project.

The Tcl file contains the following commands for each LogicLock region. Example 7–23 is for module A (instance u1) in the project named top where the region name cpll_1 was selected by Synplify Pro for the Compile Point.

Example 7–23. Commands for Each LogicLock Region in a Tcl File

```
set_global_assignment -section_id{taps_region} -name{LL_AUTO_SIZE}{ON}
set_global_assignment -section_id{taps_region} -name{LL_STATE}{FLOATING}
set_instance_assignment -section_id{taps_region} -to{|taps:u1} \
-name{LL_MEMBER_OF} {taps_region}
```

These commands create a LogicLock region with auto size and floating origin properties. This flexible LogicLock region allows the Quartus II Compiler to select the size and location of the region.



For more information about Tcl commands, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Depending on your design methodology, you can create one Quartus II project for all netlists (a top-down placement and routing flow) or a separate Quartus II project for each netlist (a bottom-up placement and routing flow). In a top-down incremental compilation design flow, you create design partition assignments and LogicLock floorplan location assignments for each partition in the design within a single Quartus II project. This methodology allows for the best quality of results and performance preservation during incremental changes to your design.

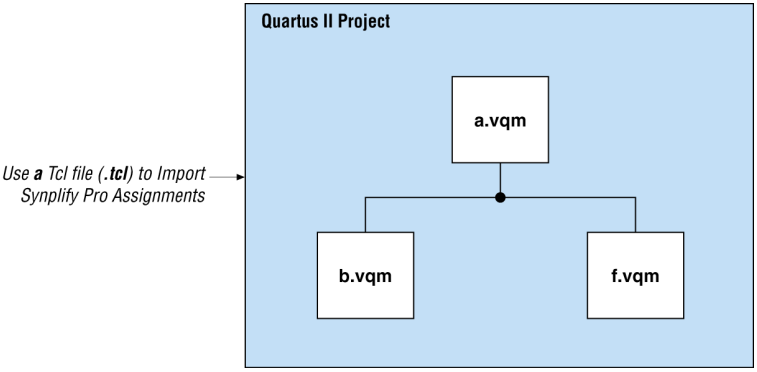
You may require a bottom-up design flow if each partition must be optimized separately, such as in certain team-based design flows. To perform a bottom-up compilation in the Quartus II software, create separate Quartus II projects and import each design partition into a top-level design using the incremental compilation export and import features to maintain placement results.

The following sections describe how to create the Quartus II projects for these two design flows.

Creating a Single Quartus II Project for a Top-Down Incremental Compilation Flow

Use the <*top-level project*>**.tcl** file that contains the Synplify Pro assignments for all partitions within the project. This method allows you to import all the partitions into one Quartus II project and optimize all modules within the project at once, taking advantage of the performance preservation and compilation-time reduction incremental compilation offers. Figure 7–4 shows a visual representation of the design flow for the example design in Figure 7–3.

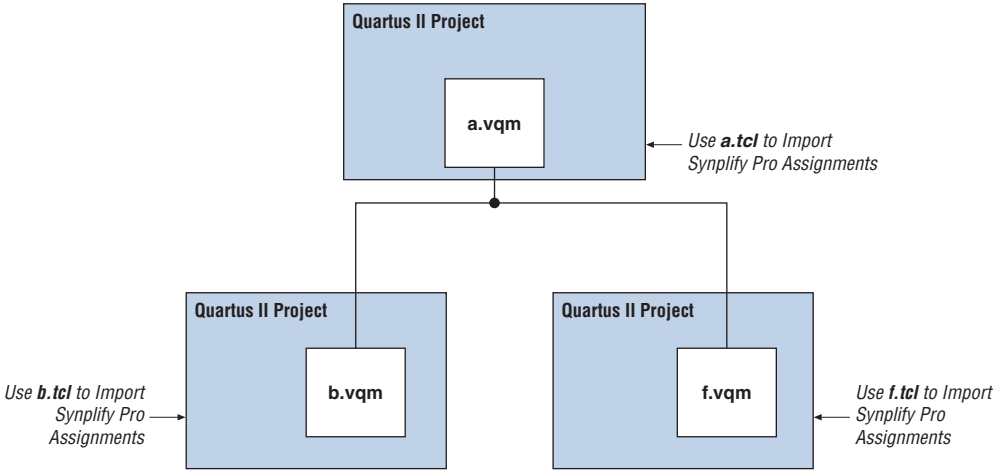
Figure 7–4. Design Flow Using Multiple VQM Files with One Quartus II Project



Creating Multiple Quartus II Projects for a Bottom-Up LogicLock Design Flow

Generate multiple Quartus II projects, one for each partition and netlist in the design. Each designer in the project can optimize their partition separately within the Quartus II software and export the placement for their partitions. Figure 7–5 shows a visual representation of the design flow for the example design in Figure 7–3. The optimized sub-designs can be brought into one top-level Quartus II project using incremental compilation.

Figure 7–5. Design Flow Using Multiple VQM Files with Multiple Quartus II Projects



Generating a Design with Multiple VQM Files Using Black Boxes

This section describes how to manually generate multiple VQM files using black boxes. This manual flow is supported in versions of the Synplify software that do not include the MultiPoint Synthesis feature.

Manually Creating Multiple VQM Files Using Black Boxes

To create multiple VQM files manually in the Synplify software, create a separate project for each low-level module and the top-level design that you want to maintain as a separate VQM file. Implement black box instantiations of lower-level partitions in your top-level project.

When synthesizing the projects for the lower-level modules, perform the following steps:

1. In the **Implementation Options** dialog box, turn on **Disable I/O Insertion** for the target technology.
2. Read the HDL files for the modules.

   Modules may include black box instantiations of lower-level modules that are also maintained as separate VQM files.
3. Add constraints with the SCOPE constraint window.
4. Enter the clock frequency to ensure that the sub-design is correctly optimized.
5. In the **Attributes** tab, set **syn_netlist_hierarchy** to **0**.

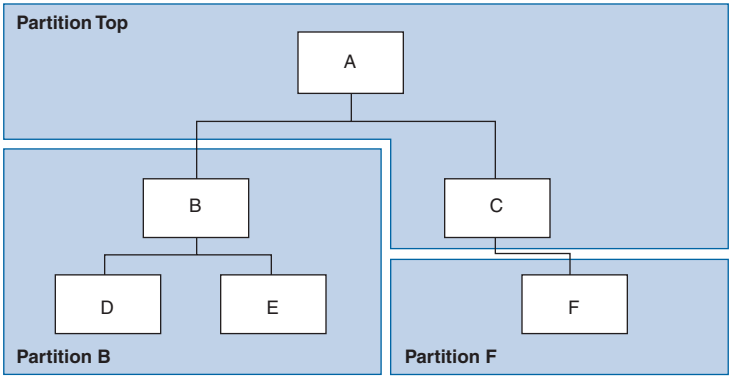
When synthesizing the top-level design project, perform the following steps:

1. Turn off **Disable I/O Insertion** for the target technology.
2. Read the HDL files for top-level designs.
3. Create black boxes using lower-level modules in the top-level design.
4. Add constraints with the SCOPE constraint window.
5. Enter the clock frequency to ensure that the design is correctly optimized.

6. In the **Attributes** tab, set **syn_netlist_hierarchy** to **0**.

The following sections describe an example of black box implementation to create separate VQM files. Figure 7–3 for an example of a design hierarchy that is split into multiple partitions.

Figure 7–6. Partitions in a Hierarchical Design



In Figure 7–3, the partition top contains the top-level block in the design (block A) and the logic that is not defined as part of another partition. In this example, the partition for top-level block A also includes the logic in one of its sub-blocks, C. Because block F is contained in its own partition, it is not treated as part of the top-level partition A. Another separate partition, B, contains the logic in blocks B, D, and E. In a team-based design, different engineers may work on the logic in different partitions. One netlist is created for the top-level module A and its submodule C, another netlist is created for B and its submodules D and E, while a third netlist is created for F. To create multiple VQM files for this design, follow these steps:

1. Generate a VQM file for module B. Use **B.v/.vhd**, **D.v/.vhd**, and **E.v/.vhd** as the source files.

2. Generate a VQM file for module F. Use **F.v/.vhd** as the source files.

3. Generate a top-level VQM file for module A. Use **A.v/.vhd** and **C.v/.vhd** as the source files. Ensure that you use black box modules B and F, which were optimized separately in the previous steps.

Creating Black Boxes in Verilog HDL

Any design block that is not defined in the project or included in the list of files to be read for a project are treated as a black box by the software. Use the `syn_black_box` attribute to indicate that you intend to create a black box for the given module. In Verilog HDL, you must provide an empty module declaration for the module that is treated as a black box.

Example 7–24 shows an example of the **A.v** top-level file. Follow the same procedure below for lower-level files which also contain a black box for any module beneath the current level hierarchy.

Example 7–24. Verilog HDL Black Box for Top-Level File A.v

```
module A (data_in, clk, e, ld, data_out);
    input data_in, clk, e, ld;
    output [15:0] data_out;

    wire [15:0] cnt_out;

    B U1 (.data_in (data_in),.clk(clk), .ld (ld),.data_out(cnt_out));
    F U2 (.d(cnt_out), .clk(clk), .e(e), .q(data_out));

    // Any other code in A.v goes here.

endmodule

// Empty Module Declarations of Sub-Blocks B and F follow here.
// These module declarations (including ports) are required for black
boxes.

module B (data_in, clk, ld, data_out) /* synthesis syn_black_box */ ;
    input data_in, clk, ld;
    output [15:0] data_out;
endmodule

module F (d, clk, e, q) / *synthesis syn_black_box */ ;
    input [15:0] d;
    input clk, e;
    output [15:0] q;
endmodule
```

Creating Black Boxes in VHDL

Any design block that is not defined in the project or included in the list of files to be read for a project are treated as a black box by the software. Use the `syn_black_box` attribute to indicate that you intend to treat the given component as a black box. In VHDL, you need a component declaration for the black box just like any other block in the design.



Although VHDL is not case-sensitive, VQM (a subset of Verilog HDL) is case-sensitive. Entity names and their port declarations are forwarded to the VQM. Black box names and port declarations are also passed to the VQM. To prevent case-based mismatches between VQM, use the same capitalization for black box and entity declarations in VHDL designs.

Example 7–25 shows an example of the **A.vhd** top-level file. Follow this same procedure for any lower-level files that contain a black box for any block beneath the current level of hierarchy.

Example 7–25. VHDL Black Box for Top-Level File A.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY synplify;
use synplify.attributes.all;

ENTITY A IS
PORT ( data_in : IN INTEGER RANGE 0 TO 15;
   clk, e, ld : IN STD_LOGIC;
   data_out : OUT INTEGER RANGE 0 TO 15 );
END A;

ARCHITECTURE a_arch OF A IS

COMPONENT B PORT(
   data_in : IN INTEGER RANGE 0 TO 15;
   clk, ld : IN STD_LOGIC;
   d_out : OUT INTEGER RANGE 0 TO 15 );
END COMPONENT;

COMPONENT F PORT(
   d : IN INTEGER RANGE 0 TO 15;
   clk, e: IN STD_LOGIC;
   q : OUT INTEGER RANGE 0 TO 15 );
END COMPONENT;

attribute syn_black_box of B: component is true;
attribute syn_black_box of F: component is true;

-- Other component declarations in A.vhd go here
```

```
signal cnt_out : INTEGER RANGE 0 TO 15;

BEGIN

U1 : B
PORT MAP (
    data_in => data_in,
    clk => clk,
    ld => ld,
    d_out => cnt_out );

U2 : F
PORT MAP (
    d => cnt_out,
    clk => clk,
    e => e,
    q => data_out );

-- Any other code in A.vhd goes here

END a_arch;
```

After you have completed the steps described in this section, you have a netlist file for each partition of the design. These files are ready for use with incremental compilation in the Quartus II software.

Creating a Quartus II Project for Multiple VQM Files

The Synplify software creates a Tcl file for each VQM file that provides the Quartus II software with the appropriate constraints and information to set up a project. For details about using the Tcl script generated by the Synplify software to set up your Quartus II project and pass your constraints, refer to "Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script" on page 7–19.

Depending on your design methodology, you can create one Quartus II project for all netlists (a top-down placement and routing flow) or a separate Quartus II project for each netlist (a bottom-up placement and routing flow). In a top-down incremental compilation design flow, you create design partition assignments and LogicLock floorplan location assignments for each partition in the design within a single Quartus II project. This methodology allows for the best quality of results and performance preservation during incremental changes to your design. You may require a bottom-up design flow where each partition must be optimized separately, such as in certain team-based design flows.

To perform a bottom-up compilation in the Quartus II software, create separate Quartus II projects and import each design partition into a top-level design using the incremental compilation export and import features to maintain placement results.

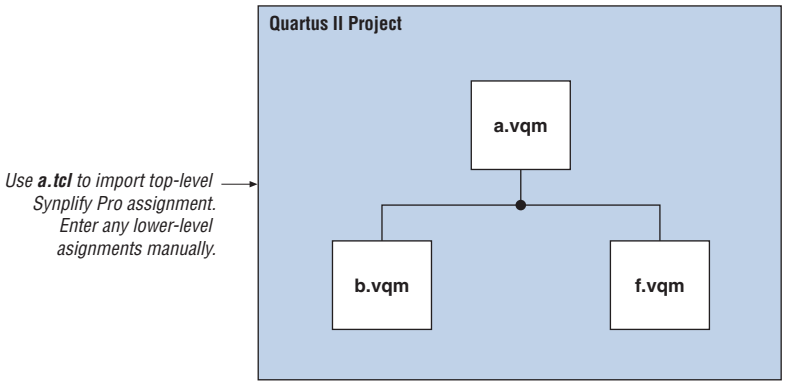
The following sections describe how to create the Quartus II projects for these two design flows.

Creating Compile Points in a Single Quartus II Project for a Top-Down Incremental Compilation Flow

Use the <*top-level project*>**.tcl** file that contains the Synplify assignments for the top-level design. This method allows you to import all of the partitions into one Quartus II project and optimize all modules within the project at once, taking advantage of the performance preservation and compilation time reduction offered by incremental compilation. Figure 7–7 shows a visual representation of the design flow for the example design in Figure 7–3.

All of the constraints from the top-level project are passed to the Quartus II software in the top-level Tcl file, but any constraints made in the lower-level projects within the Synplify software is not forward-annotated. Enter these constraints manually in your Quartus II project.

Figure 7–7. Design Flow Using Multiple VQM Files with One Quartus II Project



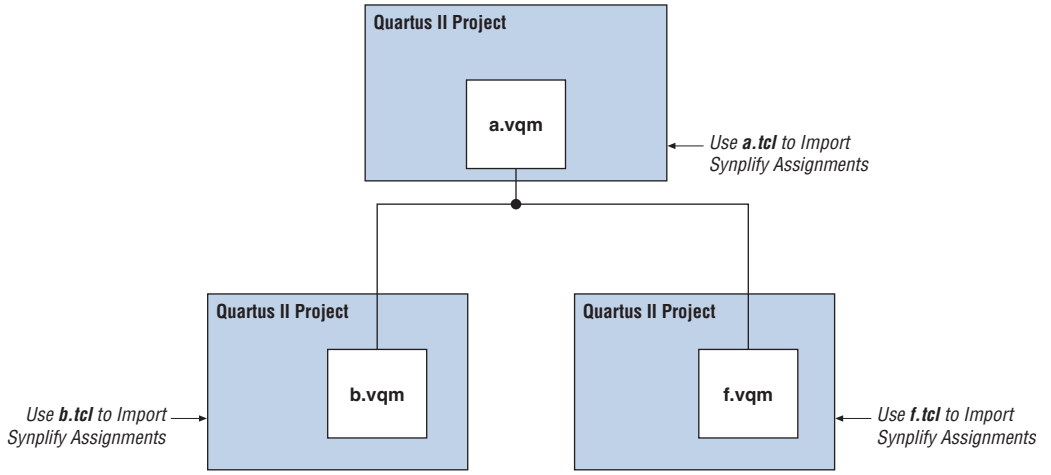
Creating Multiple Quartus II Projects for a Bottom-Up Design Flow

Use the Tcl file that is created for each VQM file by the Synplify software for each Synplify Project. This method generates multiple Quartus II projects, one for each block in the design. Each designer in the project can

optimize their block separately within the Quartus II software and export the placement of their blocks. Figure 7–8 shows a visual representation of the design flow for the example in Figure 7–3 on page 7–48.

Designers should create a LogicLock region for each block; the top-level designer should then import all the blocks and assignments into the top-level project. This method allows each block in the design to be treated separately; each block can then be imported into one top-level project.

Figure 7–8. Design Flow Using Multiple Synplify Projects and Multiple Quartus II Projects



Conclusion

Advanced synthesis is an important part of the design flow. Taking advantage of the Synplicity Synplify and Quartus II design flows allow you to control how your design files are prepared for the Quartus II place-and-route process, as well as improve performance and optimize a design for use with Altera devices. Several of the methodologies outlined in this chapter can help optimize a design to achieve performance goals and save design time.

Referenced Documents

This chapter references the following documents:

- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Tasks & Tips* chapter of the *Synplify Software User Guide*
- *Altera I/O Standards* in the *Synplify Reference Manual*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Synplify Pro User Guide and Reference Manual* on the Synplicity web site at **www.synplicity.com/literature/index.html**
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 7–6 shows the revision history for this chapter.

Table 7–6. Document Revision History

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	The following changes were made to this document: • Updated Synplicity version support • Added information on how to set the correct Quartus II version prior to compiling a MegaWizard-generated file	Updated chapter based on the Synplicity functionality supported with the Quartus II software release, version 7.2
May 2007 v7.1.0	• Removed figure 7-2 (no longer applicable) • Updated the section "Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager" to specify using the MegaWizard Plug-In Manager-generated wrapper file only when using Synplify software version 8.8 • Replaced references to "clear-box" methodologies with information supporting Synthesis Area and Timing Estimation Netlists • Minor updates for the Quartus II software version 7.1. • Added Referenced Documents section.	Updated chapter based on the Synplify software, version 8.8 and the Quartus II software release, version 7.1
March 2007 v7.0.0	• Added Cyclone III to list of devices supported • Clarified that the Synplify software generates the **.scf** file regardless of the device selected.	This chapter has been updated to include Cyclone III support for this release.

Table 7–6. Document Revision History (Continued)

Date and Version	Changes Made	Summary of Changes
November 2006 v6.1.0	• Chapter 9 was formally Chapter 8 in version 6.0.0. • Added that SCF is generated to pass SDC constraints for TimeQuest. • Added timing constraint information when using TimeQuest. • Moved note about alt_pll megafunctions from clear box section to black box section. • Clarified that Synplify reads the alt_pll megafunction black box file for Stratix and Cyclone series devices.	Updated to include Stratix III support and added information on how to pass timing constraint information for TimeQuest.
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0: • Updated cross probing information. • Added NativeLink® integration information. • Added Synplify design flow support. • Added Altera megafunction guidelines and architecture-specific features.	—
December 2005 v5.1.1	Minor typographical corrections	—
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 8 was formerly chapter 9 in version 5.0.	—
May 2005 v5.0.0	Chapter 9 was formerly chapter 7 in version 4.2.	—
December 2004 v2.1.0	• Chapter 8 was formerly Chapter 9 in version 4.1. • Updated information. • New functionality for Quartus II software version 4.2. • Updated figure 8-1.	—
June 2004 v2.0.0	• Updates to tables, figures. • New functionality for Quartus II software version 4.1.	—
February 2004 v1.0.0	Initial release.	—

8. Quartus II Integrated Synthesis

QII51008-7.2.0

Introduction

As programmable logic designs become more complex and require increased performance, advanced synthesis has become an important part of the design flow. The Quartus® II software includes advanced integrated synthesis that fully supports VHDL and Verilog HDL, as well as Altera®-specific design entry languages, and provides options to control the synthesis process. With this synthesis support, the Quartus II software provides a complete, easy-to-use solution.

This chapter documents the design flow and language support in the Quartus II software. It explains how you can use incremental compilation to reduce your compilation time, and how you can improve synthesis results with Quartus II synthesis options and by controlling the inference of architecture-specific megafunctions. This chapter also explains some of the node-naming conventions used during synthesis to help you better understand your synthesized design, and the messages issued during synthesis to improve your HDL code. Scripting techniques for applying all the options and settings described are also provided.

This chapter contains the following sections:

- "Design Flow" on page 8–2
- "Language Support" on page 8–5
- "Incremental Synthesis and Incremental Compilation" on page 8–23
- "Quartus II Synthesis Options" on page 8–24
- "Analyzing Synthesis Results" on page 8–73
- "Analyzing and Controlling Synthesis Messages" on page 8–74
- "Node-Naming Conventions in Quartus II Integrated Synthesis" on page 8–79
- "Scripting Support" on page 8–86

This chapter provides examples of how to use attributes described within the chapter, but does not cover specific coding examples.



For examples of Verilog HDL and VHDL code synthesized for specific logic functions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. For information about coding with primitives that describe specific low-level functions in Altera devices, refer to the *Designing With Low-Level Primitives User Guide*.

Design Flow

The Quartus II Analysis and Synthesis process includes Quartus II integrated synthesis, which fully supports Verilog HDL and VHDL languages as well as Altera-specific languages, and supports major features in the SystemVerilog language (refer to "Language Support" on page 8–5 for details). This stage of the compilation flow performs logic synthesis to optimize design logic, and performs technology mapping to implement the design logic using device resources, such as logic elements (LEs) or adaptive logic modules (ALMs) and other dedicated logic blocks. This stage also generates the single project database that integrates all the design files in a project (including any netlists from third-party synthesis tools).

You can use the Analysis and Synthesis stage of the Quartus II compilation to perform any of the following levels of analysis and synthesis:

- **Analyze Current File**—Parse the current design source file to check for syntax errors. This command does not report on many semantic errors that require further design synthesis. On the Processing menu, click **Analyze Current File**.
- **Analysis & Elaboration**—Check a design for syntax and semantic errors and perform elaboration to identify the design hierarchy. On the Processing menu, point to **Start**, then click **Start Analysis & Elaboration**.
- **Analysis & Synthesis**—Perform complete analysis and synthesis on a design, including technology mapping. On the Processing menu, point to **Start**, then click **Start Analysis & Synthesis**. This is the most commonly used command and is part of the full compilation flow.

The Quartus II design and compilation flow using Quartus II integrated synthesis is made up of the following steps:

1. Create a project in the Quartus II software, and specify the general project information, including the top-level design entity name. On the File menu, click **New Project Wizard**.
2. Create design files in the Quartus II software or with a text editor.
3. On the Project menu, click **Add/Remove Files in Project** and add all design files to your Quartus II project using the **Files** page of the **Settings** dialog box.
4. Specify compiler settings that control the compilation and optimization of the design during synthesis and fitting. For synthesis settings, refer to "Quartus II Synthesis Options" on page 8–24. Add timing constraints to specify the timing requirements.

5. Compile the design in the Quartus II software. To synthesize the design, on the Processing menu, point to **Start**, and click **Start Analysis & Synthesis**.

   On the Processing menu, click **Start Compilation** to run a complete compilation flow including placement, routing, creation of a programming file, and timing analysis.

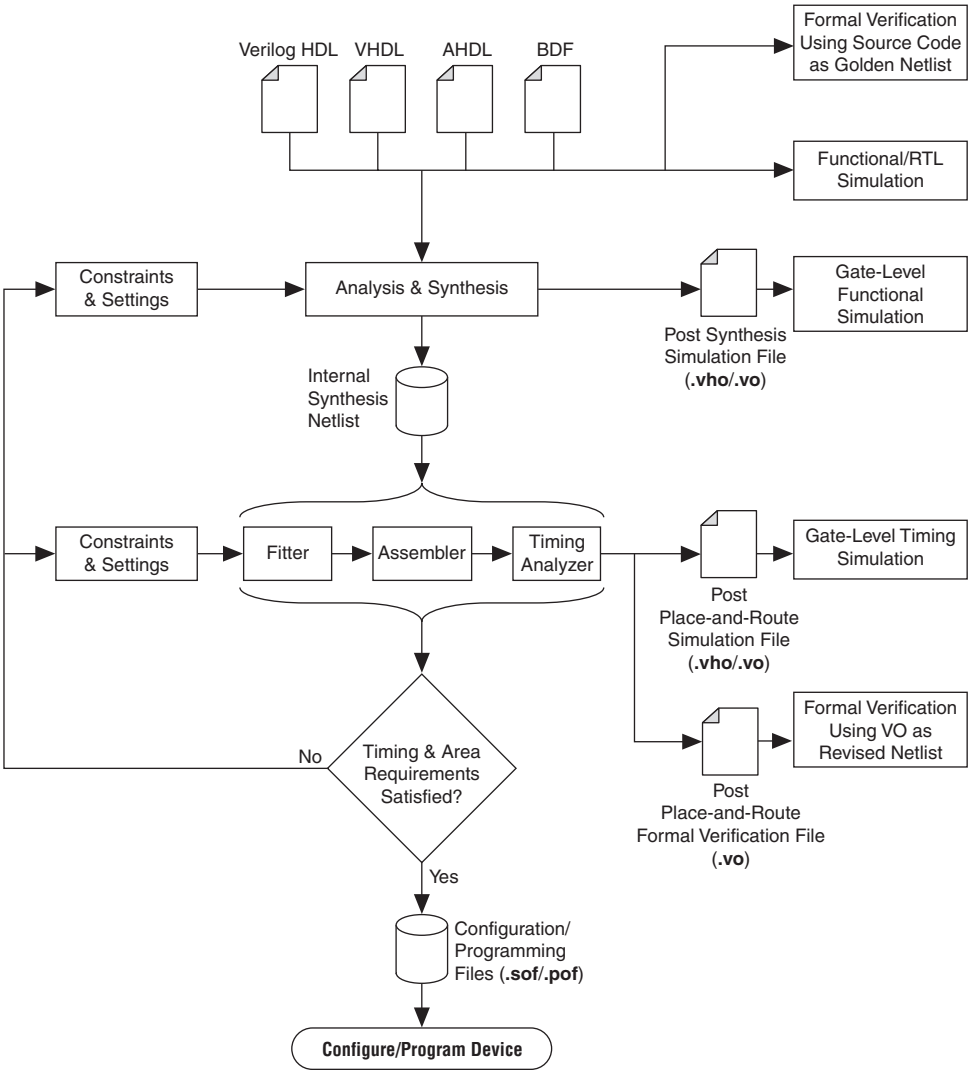
6. After obtaining synthesis and place-and-route results that meet your needs, program or configure the Altera device.

The software provides netlists that allow you to perform functional simulation and gate-level timing simulation in the Quartus II simulator or a third-party simulator, perform timing analysis in a third-party timing analysis tool in addition to the TimeQuest Timing Analyzer or Classic Timing Analyzer, and/or perform formal verification in a third-party formal verification tool. The Quartus II software also provides many additional analysis and debugging features.

For more information about creating a project, compilation flow, and other features in the Quartus II software, refer to the Quartus II Help. For an overall summary of Quartus II features, refer to the *Introduction to the Quartus II Software*.

Figure 8–1 shows the basic design flow using Quartus II integrated synthesis.

Figure 8–1. Quartus II Design Flow Using Quartus II Integrated Synthesis



Language Support

This section explains the Quartus II software's integrated synthesis support for HDL and schematic design entry, as well as graphical state machine entry, and explains how to specify the Verilog or VHDL language version used in your design. It also documents language features such as Verilog macros, initial constructs and memory system tasks, and VHDL libraries. "Design Libraries" on page 8–14 describes how to compile and reference design units in different custom libraries and "Using Parameters/Generics" on page 8–18 describes how to use parameters or generics and how to pass them between different languages.

To ensure that the software reads all associated project files, add each file to your Quartus II project. To add files to your project in the Quartus II GUI, on the Project menu, click **Add/Remove Files In Project**. Design files can be added to the project in any order. You can mix all supported languages and netlists generated by third-party synthesis tools in a single Quartus II project.

Verilog HDL Support

The Quartus II Compiler's analysis and synthesis module supports the following Verilog HDL standards:

- Verilog-1995 (IEEE Standard 1364-1995)
- Verilog-2001 (IEEE Standard 1364-2001)
- SystemVerilog-2005 (IEEE Standard 1800-2005) (not all constructs are supported)



For complete information about specific Verilog syntax features and language constructs, refer to the Quartus II Help.

The Quartus II Compiler uses the Verilog-2001 standard by default for files that have the extension **.v**, and the SystemVerilog standard for files that have the extension **.sv**.



The Verilog HDL code samples provided in this document follow the Verilog-2001 standard unless otherwise specified.

To specify a default Verilog HDL version for all files, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Settings** dialog box, under Category, expand **Analysis & Synthesis Settings**, and select **Verilog HDL Input**.

3. On the **Verilog HDL Input** page, under **Verilog version**, select the appropriate Verilog version, then click **OK**.

You can override the default Verilog HDL version for each Verilog design file by performing the following steps:

1. On the Project menu, click **Add/Remove Files in Project**. The **Settings** dialog box appears.
2. On the **Files** page, click on the appropriate file in the list and click the **Properties** button.
3. In the **HDL Version** list, select **SystemVerilog_2005**, **Verilog_2001**, or **Verilog_1995** and click **OK**.

You can also control the Verilog HDL version inside a design file using the `VERILOG_INPUT_VERSION` synthesis directive, as shown in Example 8–1. This directive overrides the default HDL version and any HDL version specified in the **File Properties** dialog box.

Example 8–1. Controlling the Verilog HDL Input Version with a Synthesis Directive

```
// synthesis VERILOG_INPUT_VERSION <language version>
```

The variable <*language version*> takes one of the following values:

- `VERILOG_1995`
- `VERILOG_2001`
- `SYSTEMVERILOG_2005`

When the software reads a `VERILOG_INPUT_VERSION` synthesis directive, the current language version changes as specified until the end of the file, or until the next `VERILOG_INPUT_VERSION` directive is reached.



You cannot change the language version in the middle of a Verilog module.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 8–29.

If you use scripts to add design files, you can use the `-HDL_VERSION` command to specify the HDL version for each design file. Refer to "Adding an HDL File to a Project and Setting the HDL Version" on page 8–87.

The Quartus II software support for Verilog HDL is case-sensitive in accordance with the Verilog HDL standard. The Quartus II software supports the compiler directive `define, in accordance with the Verilog HDL standard.

The Quartus II software supports the include compiler directive to include files with absolute paths (with either "/" or "\" as the separator), or relative paths (relative to project root, user libraries, or current file location). When searching for a relative path, the Quartus II software initially searches relative to the project directory. If the software cannot find the file, it then searches relative to all user libraries, and finally relative to the directory location of the current file.

Verilog-2001 Support

The Quartus II software does not support Verilog-2001 libraries and configurations.

SystemVerilog Support

The Quartus II software supports the following SystemVerilog constructs:

- Parameterized interfaces, generic interfaces, and modport constructs
- Packages
- Extern module declarations
- Built-in data types logic, bit, byte, shortint, longint, int
- Unsized integer literals '0, '1, 'x, 'z, 'X, and 'Z
- Structure data types using struct
- Ports and parameters with unrestricted data types
- Unpacked and packed arrays (does not support packed arrays with more than one dimension)
- User-defined types using typedef
- Global declarations of task/functions/parameters/types (does not support global variables)
- Coding constructs always_comb, always_latch, always_ff
- Continuous assignments to nodes other than nets, and procedural assignments to nodes other than reg
- Enumeration methods First, Last, Next(n), Prev(n), Num, and Name
- Assignment operators +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, and >>>=
- Increment ++ and decrement --
- Jump statements return, break, and continue
- Enhanced for loop (declare loop variables inside initial condition)
- Do-while loop

- Assignment patterns
- Keywords `unique` and `priority` in case statements
- Default values for function/task arguments
- Closing labels
- Extensions to directives `` `define `` and `` `include ``
- Expression size system function `$bits`
- Array query system functions `$dimensions`, `$unpacked_dimensions`, `$left`, `$right`, `$high`, `$low`, `$increment`, `$size`

Quartus II integrated synthesis also parses, but otherwise ignores, SystemVerilog assertions.



Designs written to comply with the Verilog-2001 standard may not compile successfully using the SystemVerilog setting because the SystemVerilog standard adds a number of new reserved keywords. For a list of reserved words in each language standard, refer to the Quartus II Help.

Initial Constructs and Memory System Tasks

The Quartus II software infers power-up conditions from Verilog `initial` constructs. The software creates power-up settings for variables, including RAM blocks. If the Quartus II software encounters non-synthesizable constructs in an `initial` block, it generates an error. To avoid such errors, enclose non-synthesizable constructs (such as those intended only for simulation) in `translate_off` and `translate_on` synthesis directives, as described in "Translate Off and On / Synthesis Off and On" on page 8–65. Synthesis of initial constructs enables the power-up state of the synthesized design to match, as closely as possible, the power-up state of the original HDL code in simulation.



Initial blocks do not infer power-up conditions in some third-party EDA synthesis tools. If converting between synthesis tools, ensure that your power-up conditions are set correctly.

Quartus II integrated synthesis supports the `$readmemb` and `$readmemh` system tasks to initialize memories. Example 8–2 shows an initial construct that initializes an inferred RAM with `$readmemb`.

Example 8–2. Verilog Example of Initializing RAM with the readmemb Command

```
reg [7:0] ram[0:15];
initial
begin
  $readmemb("ram.txt", ram);
end
```

When creating a text file to use for memory initialization, specify the address using the format @<*location*> on a new line, then specify the memory word such as `110101` or `abcde` on the next line. Example 8–3 shows a portion of a memory initialization file for the RAM in Example 8–2.

Example 8–3. Text File Format for Initializing RAM with the readmemb Command

```
@0
00000000
@1
00000001
@2
00000010
…
@e
00001110
@f
00001111
```

Verilog HDL Macros

The Quartus II software fully supports Verilog HDL macros, which you can define with the `` `define `` compiler directive in your source code. You can also define macros in the GUI or on the command line.

Setting a Verilog Macro Default Value in the GUI
To specify a macro in the GUI, on the Assignments menu, click **Settings**. Under **Category**, expand **Analysis & Synthesis Settings** and click **Verilog HDL Input**. Under **Verilog HDL macro**, type the macro name in the **Name** box, the value in the **Setting** box, and click **Add**.

Setting a Verilog Macro Default Value on the Command Line
To set a default value for a Verilog macro on the command line, use the `--verilog_macro` option, as shown in Example 8–4.

Example 8–4. Command Syntax for Specifying a Verilog Macro

```
quartus_map <Design name> --verilog_macro= "<Macro Name>=<Macro Setting>" ↵
```

The command in Example 8–5 has the same effect as specifying `` `define a 2 `` in the Verilog HDL source code.

Example 8–5. Specifying a Verilog Macro a = 2

```
quartus_map my_design --verilog_macro="a=2" ↵
```

To specify multiple macros, you can repeat the option more than once, as in Example 8–6.

Example 8–6. Specifying Verilog Macros a = 2 and a = 3

```
quartus_map my_design --verilog_macro="a=2" --verilog_macro="b=3" ↵
```

VHDL Support

The Quartus II Compiler's Analysis and Synthesis module supports the following VHDL standards:

- VHDL 1987 (IEEE Standard 1076-1987)
- VHDL 1993 (IEEE Standard 1076-1993)

For information about specific VHDL syntax features and language constructs, refer to the Quartus II Help.

The Quartus II Compiler uses the VHDL 1993 standard by default for files that have the extension **.vhdl** or **.vhd**.

The VHDL code samples provided in this document follow the VHDL 1993 standard.

To specify a default VHDL version for all files, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Settings** dialog box, under **Category**, expand **Analysis & Synthesis Settings**, and select **VHDL Input**.
3. On the **VHDL Input** page, under **VHDL version**, select the appropriate version, then click **OK**.

You can override the default VHDL version for each VHDL design file by performing the following steps:

1. On the Project menu, click **Add/Remove Files in Project**. The **Settings** dialog box appears.
2. On the **Files** page, click on the appropriate file in the list and click **Properties**.
3. In the HDL version list, select **VHDL93** or **VHDL87** and click **OK**.

You can also specify the VHDL version for each design file using the `VHDL_INPUT_VERSION` synthesis directive, as shown in Example 8–7. This directive overrides the default HDL version and any HDL version specified in the **File Properties** dialog box.

Example 8–7. Controlling the VHDL Input Version with a Synthesis Directive

```
--synthesis VHDL_INPUT_VERSION <language version>
```

The variable <*language version*> takes one of the following values:

- `VHDL87`
- `VHDL93`

When the software reads a `VHDL_INPUT_VERSION` synthesis directive, it changes the current language version as specified until the end of the file, or until it reaches the next `VHDL_INPUT_VERSION` directive.



You cannot change the language version in the middle of a VHDL design unit.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 8–29.

If you use scripts to add design files, you can use the –`HDL_VERSION` command to specify the HDL version for each design file. Refer to "Adding an HDL File to a Project and Setting the HDL Version" on page 8–87.

The Quartus II software reads default values for registered signals defined in the VHDL code and converts the default values into power-up level settings. This enables the power-up state of the synthesized design to match, as closely as possible, the power-up state of the original HDL code in simulation.

VHDL Standard Libraries and Packages

The Quartus II software includes the standard IEEE libraries and a number of vendor-specific VHDL libraries. For information about organizing your own design units into custom libraries, refer to "Design Libraries" on page 8–14.

The **IEEE** library includes the standard VHDL packages `std_logic_1164`, `numeric_std`, `numeric_bit`, and `math_real`. The **STD** library is part of the VHDL language standard and includes the packages `standard` (included in every project by default) and `textio`. For compatibility with older designs, the Quartus II software also supports the following vendor-specific packages and libraries:

- Synopsys packages such as `std_logic_arith` and `std_logic_unsigned` in the **IEEE** library
- Mentor Graphics® packages such as `std_logic_arith` in the **ARITHMETIC** library
- Altera primitive packages `altera_primitives_components` (for primitives such as `GLOBAL` and `DFFE`) and `maxplus2` (for legacy support of MAX+PLUS® II primitives) in the **ALTERA** library
- Altera megafunction packages `altera_mf_components` and `stratixgx_mf_components` in the **ALTERA_MF** library (for Altera-specific megafunctions including LCELL), and `lpm_components` in the **LPM** library for library of parameterized modules (LPM) functions.

For a complete listing of library and package support, refer to the Quartus II Help.

Beginning with the Quartus II software version 5.1, you should import component declarations for Altera primitives such as `GLOBAL` and `DFFE` from the `altera_primitives_components` package and not the `altera_mf_components` package.

AHDL Support

The Quartus II Compiler's Analysis and Synthesis module fully supports the Altera Hardware Description Language (AHDL).

AHDL designs use Text Design Files (**.tdf**). You can import AHDL Include Files (**.inc**) into a Text Design File with an AHDL `include` statement. Altera provides AHDL Include Files for all megafunctions shipped with the Quartus II software.



For information about specific AHDL syntax features and language constructs, refer to the Quartus II Help.



The AHDL language does not support the synthesis directives or attributes described in this chapter.

Schematic Design Entry Support

The Quartus II Compiler's analysis and synthesis module fully supports Block Design Files (**.bdf**) for schematic design entry.

You can use the Quartus II software's Block Editor to create and edit Block Design Files and open Graphic Design Files (**.gdf**) imported from the MAX+PLUS II software. Use the Symbol Editor to create and edit Block Symbol Files (**.bsf**) and open MAX+PLUS II Symbol Files (**.sym**). You can read and edit these legacy MAX+PLUS II formats with the Quartus II Block and Symbol Editors; however, the Quartus II software saves them as **.bdf** or **.bsf** files.



For information about creating and editing schematic designs, refer to the Quartus II Help.



Schematic entry methods do not support the synthesis directives or attributes described in this chapter.

State Machine Editor

The Quartus II software supports graphical state machine entry. To create a new finite state machine (FSM) design, on the File menu, click **New**. On the **Device Design Files** tab, choose **State Machine Editor.**

In the editor, you can use the State Machine Wizard to step you through the state machine creation. Click the **State Machine Wizard** icon. Specify the reset information, define the input ports, states, and transitions, and then define the output ports and output conditions. Click **Finish** to create the state machine diagram.

Alternately, create the state machine diagram in the editor GUI. Use the icons or right-click menu options to insert new input and output signals and create states in the schematic display. To specify transitions, select the **Transition Tool** and click on the source state, then drag the mouse to the destination state. Double-click on a transition to specify the transition equation, using a syntax that conforms to Verilog HDL. Double-click on a state to open the **State Properties** dialog box, where you can change the state name, specify whether it acts as the reset state, and change the incoming and outgoing transition equations.

To view and edit state machine information in a table format, click the **State Machine Table** icon.

The state machine diagram is saved as a State Machine File (**.smf**). When you have finished defining the state machine logic, create a Verilog HDL or VHDL design file by clicking the **Generate HDL File** icon. You can then instantiate the state machine in your design using any design entry language.



For more information about creating and editing state machine diagrams, refer to the Quartus II Help.

Design Libraries

By default, the Quartus II software compiles all design files into the work library. If you do not specify a design library, or if a file refers to a library that does not exist, or if the library does not contain a referenced design unit, the software searches the work library. This behavior allows the Quartus II software to compile most designs with minimal setup. (Creating separate custom design libraries is optional.)

To compile your design files into specific libraries (for example, when you have two or more functionally different design entities that share the same name), you can specify a destination library for each design file in various ways, as described in the following subsections:

- "Specifying a Destination Library Name in the Settings Dialog Box" on page 8–15
- "Specifying a Destination Library Name in the Quartus II Settings File or Using Tcl" on page 8–15

When the Quartus II Compiler analyzes the file, it stores the analyzed design units in the file's destination library.



Beginning with the Quartus II software version 6.1, a design can contain two or more entities with the same name if they are compiled into separate libraries.

When compiling a design instance, the Quartus II software initially searches for the entity in the library associated with the instance (which is the work library if no other library is specified). If the entity definition is not found, the software searches for a unique entity definition in all design libraries. If more than one entity with the same name is found, the software generates an error. If your design uses multiple entities with the same name, you must compile the entities into separate libraries.

In VHDL, there are several ways to associate an instance with a particular entity, as described in "Mapping a VHDL Instance to an Entity in a Specific Library" on page 8–16. In Verilog HDL, BDF, AHDL, as well as VQM and EDIF netlists, use different libraries for each of the entities that have the same name, and compile the instantiation into the same library as the appropriate entity.

Specifying a Destination Library Name in the Settings Dialog Box

To specify a library name for one of your design files, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. On the **Files** page of the **Settings** dialog box, select the file in the **File Name** list.
3. Click **Properties**.
4. In the **File Properties** dialog box, select the type of design file from the **Type** list.
5. Type the desired library name in the **Library** field.
6. Click **OK**.

Specifying a Destination Library Name in the Quartus II Settings File or Using Tcl

You can specify the library name with the `-library` option to the *<language type>*`_FILE` assignment in the Quartus II Settings File or with Tcl commands.

For example, the following Quartus II Settings File or Tcl assignments specify that the Quartus II software analyze **my_file.vhd** and store its contents (design units) in the VHDL library **my_lib**, and analyze the Verilog file **my_header_file.h** and store its contents in a library called **another_lib**.

Example 8–8. Specifying a Destination Library Name

```
set_global_assignment -name VHDL_FILE my_file.vhd -library my_lib
set_global_assignment -name VERILOG_FILE my_header_file.h -library\
another_lib
```

For more information about Tcl scripting, refer to "Scripting Support" on page 8–86.

Specifying a Destination Library Name in a VHDL File

You can use the `library` synthesis directive to specify a library name in your VHDL source file. This directive takes a single string argument: the name of the destination library. Specify the `library` directive in a VHDL comment prior to the context clause for a primary design unit (that is, a package declaration, an entity declaration, or a configuration), using one of the supported keywords for synthesis directives, that is, `altera`, `synthesis`, `pragma`, `synopsys`, or `exemplar`.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 8–29.

The `library` directive overrides the default library destination **work**, the library setting specified for the current file through the **Settings** dialog box, any existing Quartus II Settings File setting, any setting made through the Tcl interface, or any prior `library` directive in the current file. The directive remains effective until the end of the file or the next `library` synthesis directive.

Example 8–9 uses the `library` synthesis directive to create a library called **my_lib** that contains the design unit `my_entity`.

Example 8–9. Using the library Synthesis Directive

```
-- synthesis library my_lib
library ieee;
use ieee.std_logic_1164.all;
entity my_entity(...)
end entity my_entity;
```



You can specify a single destination library for all the design units in a given source file by specifying the library name in the the **Settings** dialog box, editing the Quartus II Settings File, or using the Tcl interface. Using the `library` directive to change the destination VHDL library within a source file gives you the option of organizing the design units in a single file into different libraries, rather than just a single library.

The Quartus II software produces an error if you use the library directive within a design unit.

Mapping a VHDL Instance to an Entity in a Specific Library

The VHDL language provides a number of ways to map or bind an instance to an entity in a specific library, as described in the following subsections.

Direct Entity Instantiation

In the direct entity instantiation method, the instantiation refers to an entity in a specific library, as shown in Example 8–10.

Example 8–10. VHDL Example of Direct Entity Instantiation

```
entity entity1 is
port(...);
end entity entity1;

architecture arch of entity1 is
begin
   inst: entity lib1.foo
   port map(...);
end architecture arch;
```

Component Instantiation—Explicit Binding Indication

There is more than one mechanism for binding a component to an entity. In an explicit binding indication, you bind a component instance to a specific entity, as shown in Example 8–11.

Example 8–11. VHDL Example of Explicit Binding Instantiation

```
entity entity1 is
port(...);
end entity entity1;

package components is
   component entity1 is
      port map (...);
   end component entity1;
end package components;

entity top_entity is
port(...);
end entity top_entity;

use lib1.components.all;
architecture arch of top_entity is
-- Explicitly bind instance I1 to entity1 from lib1
for I1: entity1 use
   entity lib1.entity1
     port map(...);
begin
   I1: entity1 port map(...);
end architecture arch;
```

Component Instantiation—Default Binding
If you do not provide an explicit binding indication, a component instance is bound to the nearest visible entity with the same name. If no such entity is visible in the current scope, the instance is bound to the entity in the library in which the component was declared. For example, if the component is declared in a package in library MY_LIB, an instance of the component is bound to the entity in library MY_LIB. The portions of code in Example 8–12 and 8–13 show this instantiation method.

Example 8–12. VHDL Example of Default Binding to the Entity in the Same Library as the Component Declaration

```
use mylib.pkg.foo; -- import component declaration from package "pkg" in
                   -- library "mylib"
architecture rtl of top
...
begin
    -- This instance will be bound to entity "foo" in library "mylib"
    inst: foo
    port map(...);
end architecture rtl;
```

Example 8–13. VHDL Example of Default Binding to the Directly Visible Entity

```
use mylib.foo; -- make entity "foo" in library "mylib" directly visible
architecture rtl of top
    component foo is
        generic (…)
        port ( );
    end component;
begin
    -- This instance will be bound to entity "foo" in library "mylib"
    inst: foo
    port map(...);
end architecture rtl;
```

Using Parameters/Generics

This section describes how parameters (called generics in VHDL) are supported in the Quartus II software, and how you can pass these parameters between different design languages.

You can enter default parameter values for your design in the **Default Parameters** box in the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Default parameters allow you to specify the parameter overrides for your top-level entity. In AHDL, parameters are inherited, so any default parameters apply to all AHDL instances in the design. You can also specify parameters for instantiated modules in a

Block Design File. To modify parameters on a BDF instance, double-click on the parameter value box for the instance symbol, or right-click on the symbol and choose **Properties**, then click the **Parameters** tab. For these GUI-based entry methods, refer to "Setting Default Parameter Values and BDF Instance Parameter Values" on page 8–19 for information about how parameter values are interpreted, and for recommendations about the format you should use.

You can specify parameters for instantiated modules in your design source files, using the syntax provided for that language. Some designs instantiate entities in a different language; for example, they may instantiate a VHDL entity from a Verilog design file. You can pass parameters or generics between VHDL, Verilog HDL, AHDL, and BDF schematic entry, and from EDIF or VQM to any of these languages. In most cases, you do not have to do anything special to pass parameters from one language to another. However, in some cases you may have to specify the type of the parameter you are passing. In those cases you should follow certain guidelines to ensure that the parameter value is interpreted correctly. Refer to "Passing Parameters Between Two Design Languages" on page 8–20 for parameter type rules.

Setting Default Parameter Values and BDF Instance Parameter Values

Default parameter values and BDF instance parameter values do not have an explicitly declared type. In most cases, the Quartus II software can correctly infer the type from the value without ambiguity. For example, "`ABC`" is interpreted as a string, `123` as an integer, and `15.4` as a floating-point value. In other cases, such as when the instantiated subdesign language is VHDL, the Quartus II software uses the type of the parameter/generic in the instantiated entity to determine how to interpret the value, so that a value of `123` is interpreted as a string if the VHDL parameter is of type string. In addition, you can set the parameter value in a format that is legal in the language of the instantiated entity. For example, to pass an unsized bit literal value from BDF to System/Verilog, you can use `'1` as the parameter value, and to pass a 4-bit binary vector from BDF to Verilog, you can use `4'b1111` as the parameter value.

In a few cases, the Quartus II software cannot infer the correct type of parameter value. To avoid ambiguity, specify the parameter value in a type-encoded format where the first or first and second character of the parameter indicate the type of the parameter, and the rest of the string indicates the value in a quoted sub-string. For example, to pass a binary string `1010` from BDF to Verilog HDL, you cannot simply use the value `1001`, because the Quartus II software interprets it as a decimal value. You also cannot use the string `"1001"`, because the Quartus II software interprets it as an ASCII string. You must use the type-encoded string `B"1001"` for the Quartus II software to interpret the parameter value

correctly. Table 8–1 provides a list of valid parameter strings and shows how they are interpreted within the Quartus II software. Altera recommends using the type-encoded format only when necessary to resolve ambiguity.

Table 8–1. Valid Parameter Strings and How They are Interpreted

Parameter String	Quartus II Parameter Type, Format, and Value
S"abc", s"abc"	String value “abc”
"abc123", "123abc"	String value abc123 or 123abc
F"12.3", f"12.3"	Floating point number 12.3
-5.4	Floating point number -5.4
D"123", d"123"	Decimal number 123
123, -123	Decimal number 123, -123
X"ff", H"ff"	Hexadecimal value FF
Q"77", O"77"	Octal value 77
B"1010", b"1010"	Unsigned binary value 1010
SB"1010", sb"1010"	Signed binary value 1010
R"1", R"0", R"X", R"Z", r"1", r"0", r"X", r"Z"	Unsized bit literal
E"apple", e"apple"	Enum type, value name is apple
P"1 unit"	Physical literal, the value is (1, unit)
A(…), a(…)	Array type or record type, whose content is determined by the string (...)

Passing Parameters Between Two Design Languages

When passing a parameter between two different languages, a design block that is higher in the design hierarchy instantiates a lower-level subdesign block and provides parameter information. It is essential that the parameter be correctly interpreted by the subdesign language (the design entity that is instantiated). Based on the information provided by the higher-level design and the value format, and sometimes by the parameter type of the subdesign entity, the Quartus II software interprets the type and value of the passed parameter.

When passing a parameter whose value is an enumerated type value or literal from a language that does not support enumerated types to one that does (for example from Verilog to VHDL), it is essential that the enumeration literal is spelled correctly in the higher-level design. The parameter value is passed as a string literal, and it is up to the language of the lower-level design to correctly convert the string literal into the correct enumeration literal.

If the lower-level language is SystemVerilog, it is essential that the enum value is used in the correct case. In SystemVerilog, it is recommended that two enumeration literals do not only differ in case. For example, `enum {item, ITEM}` is not a good choice of item names because these names can create confusion among users and it is more difficult to pass parameters from case-insensitive HDLs, such as VHDL.

Arrays have different support in different design languages. For details about the array parameter format, refer to the **Parameter** section in the Analysis & Synthesis Report of a design that contains array parameters or generics.

The following code shows examples of passing parameters from one design entry language to a subdesign written in another language. Example 8–14 shows a VHDL subdesign that is instantiated into a top-level Verilog design in Example 8–15. Example 8–16 shows a Verilog subdesign that is instantiated in a top-level VHDL design in Example 8–17.

Example 8–14. VHDL Parameterized Subdesign Entity

```
type fruit is (apple, orange, grape);
entity vhdl_sub is
   generic (
      name : string := "default",
      width : integer := 8,
      number_string : string := "123",
      f : fruit := apple,
      binary_vector : std_logic_vector(3 downto 0) := "0101",
      signed_vector : signed (3 downto 0) := "1111");
```

Example 8–15. Verilog HDL Top-level Design Instantiating and Passing Parameters to VHDL Entity from Example 8–14

```
vhdl_sub inst (...);
      defparam inst.name = "lower";
      defparam inst.width = 3;
      defparam inst.num_string = "321";
      defparam inst.f = "grape"; // Must exactly match enum value
      defparam inst.binary_vector = 4'b1010;
      defparam inst.signed_vector = 4'sb1010;
```

Example 8–16. Verilog HDL Parameterized Subdesign Module

```
module veri_sub (...)
parameter name = "default";
parameter width = 8;
parameter number_string = "123";
parameter binary_vector = 4'b0101;
parameter signed_vector = 4'sb1111;
```

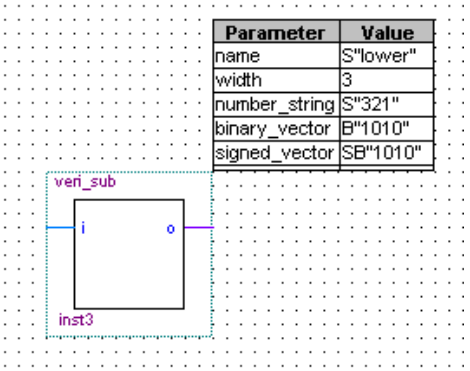
Example 8–17. VHDL Top-level Design Instantiating and Passing Parameters to the Verilog Module from Example 8–16

```
inst:veri_sub
   generic map (
      name => "lower",
      width => 3,
      number_string => "321"
      binary_vector = "1010"
      signed_vector = "1010")
```

To use an HDL subdesign such as the one shown in Example 8–16 in a top-level BDF design, you must first generate a symbol for the HDL file, as shown in Figure 8–2. Open the HDL file in the Quartus II software, and then, on the File menu, point to **Create/Update** and click **Create Symbol Files for Current File**.

To modify parameters on a BDF instance, double-click on the parameter value box for the instance symbol, or right-click on the symbol and choose **Properties**, then click the **Parameters** tab.

Figure 8–2. BDF Top-Level Design Instantiating and Passing Parameters to the Verilog Module from Example 8–16.



Incremental Synthesis and Incremental Compilation

The incremental compilation feature in the Quartus II software manages a design hierarchy for incremental design by allowing you to divide the design into multiple partitions. Incremental compilation ensures that when a design is compiled, only those partitions of the design that have been updated will be resynthesized, reducing compilation time and runtime memory usage. This also means that node names are maintained during synthesis for all registered and combinational nodes in unchanged partitions.

You can use just incremental synthesis, or use the default full incremental compilation flow in which you can also preserve the placement (and optionally routing) information for unchanged partitions. This feature allows you to preserve performance of unchanged blocks in your design and reduces the time required for placement and routing, which significantly reduces your design compilation time. Altera recommends using the full incremental compilation feature even if you want to preserve just the synthesis information. You can perform incremental synthesis by using full incremental compilation with the Netlist Type for all design partitions set to Post-Synthesis. Some Quartus II features, such as formal verification and incremental SignalTap® II logic analysis, require that the full incremental compilation feature be turned on.



For information about using the recommended full incremental compilation flow, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For information about the Incremental synthesis only option, refer to the Quartus II Help.

Partitions for Preserving Hierarchical Boundaries

A design partition represents a portion of the design that you want to synthesize and fit incrementally. Incremental compilation maintains the hierarchical boundaries of design partitions, so you can use design partitions if you need to preserve hierarchical boundaries through the synthesis and fitting process. For example, if you are performing formal verification, you must use partitions with the full incremental compilation flow to ensure that no optimizations occur across specific design hierarchies.



Beginning with the Quartus II software version 6.0, Altera recommends that you use Design Partition assignments instead of the **Preserve Hierarchical Boundary** logic option, which may be removed in future versions of the Quartus II software.

Quartus II Synthesis Options

The Quartus II software offers a number of options to help you control the synthesis process and achieve the optimal results for your design. "Setting Synthesis Options" on page 8–25 describes the **Analysis & Synthesis Settings** page of the **Settings** dialog box, where you can set the most common global settings and options, and defines the following three types of synthesis options: Quartus II logic options, synthesis attributes, and synthesis directives. The other subsections describe the following common synthesis options in the Quartus II software, and provide HDL examples of how to use each option where applicable:

- Major Optimization Settings
  - "Optimization Technique" on page 8–30
  - "Speed Optimization Technique for Clock Domains" on page 8–30
  - "PowerPlay Power Optimization" on page 8–31
  - "Restructure Multiplexers" on page 8–32
- State Machine Settings and Enumerated Types
  - "State Machine Processing" on page 8–34
  - "Manually Specifying State Assignments Using the syn_encoding Attribute" on page 8–35
  - "Manually Specifying Enumerated Types Using the enum_encoding Attribute" on page 8–38
  - "Safe State Machines" on page 8–40
- Register Power-Up Settings
  - "Power-Up Level" on page 8–42
  - "Power-Up Don't Care" on page 8–43
- Controlling, Preserving, Removing, and Duplicating Logic and Registers
  - "Remove Duplicate Registers" on page 8–44
  - "Remove Redundant Logic Cells" on page 8–44
  - "Preserve Registers" on page 8–44
  - "Disable Register Merging/Don't Merge Register" on page 8–45
  - "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 8–46
  - "Keep Combinational Node/Implement as Output of Logic Cell" on page 8–47
  - "Don't Retime, Disabling Synthesis Netlist Optimizations" on page 8–48
  - "Don't Replicate, Disabling Synthesis Netlist Optimizations" on page 8–49
  - "Maximum Fan-Out" on page 8–50
  - "Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable" on page 8–51

- To preserve design hierarchy, refer to "Partitions for Preserving Hierarchical Boundaries" on page 8–23

- Megafunction Inference Options
  - "Megafunction Inference Control" on page 8–52
  - "RAM Style and ROM Style—for Inferred Memory" on page 8–55
  - "Turning Off Add Pass-Through Logic to Inferred RAMs/ no_rw_check Attribute Setting" on page 8–57
  - "RAM Initialization File—for Inferred Memory" on page 8–59
  - "Multiplier Style—for Inferred Multipliers" on page 8–59

- Controlling Synthesis with Other Synthesis Directives
  - "Full Case" on page 8–62
  - "Parallel Case" on page 8–63
  - "Translate Off and On / Synthesis Off and On" on page 8–65
  - "Ignore translate_off and synthesis_off Directives" on page 8–65
  - "Read Comments as HDL" on page 8–66

- Specifying I/O-Related Assignments
  - "Use I/O Flipflops" on page 8–67
  - "Specifying Pin Locations with chip_pin" on page 8–68

- Setting Quartus II Logic Options in Your HDL Source Code
  - "Using altera_attribute to Set Quartus II Logic Options" on page 8–70

Setting Synthesis Options

You can set synthesis options in the **Settings** dialog box, or with logic options in the Quartus II software, or you can use synthesis attributes and directives within the HDL source code.

Analysis & Synthesis Settings Page of the Settings Dialog Box

On the Assignments menu, click **Settings** to open the **Settings** dialog box. The **Analysis & Synthesis Settings** page allows you to set global synthesis options that apply to the entire project. These options are described in later subsections.

Quartus II Logic Options

Quartus II logic options control many aspects of the synthesis and place-and-route process. To set logic options in the Quartus II graphical user interface, on the Assignments menu, click **Assignment Editor**. You can also use a corresponding Tcl command. Quartus II logic options allow

you to set instance or node-specific assignments without editing the source HDL code. Logic options can be used with all design entry languages supported by the Quartus II software.

For more information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

Synthesis Attributes

The Quartus II software supports synthesis attributes for Verilog HDL and VHDL, also commonly called pragmas. These attributes are not standard Verilog HDL or VHDL commands; synthesis tools use attributes to control the synthesis process in a particular manner. Attributes always apply to a specific design element, and are applied in the HDL source code. Some synthesis attributes are also available as Quartus II logic options via the Quartus II user interface or with Tcl. Each attribute description in this chapter indicates whether there is a corresponding setting or logic option that can be set in the user interface; some attributes can be specified only with HDL synthesis attributes.

Attributes specified in your HDL code are not visible in the Assignment Editor or in the Quartus II Settings File. Assignments or settings made through the Quartus II user interface, the Quartus II Settings File, or the Tcl interface take precedence over assignments or settings made with synthesis attributes in your HDL code. The Quartus II software generates warning messages if invalid attributes are found, but does not generate an error or stop the compilation. This behavior is required because attributes are specific to various design tools, and attributes not recognized in the Quartus II software may be intended for a different EDA tool. The Quartus II software lists the attributes specified in your HDL code in the **Source assignments** table in the Analysis & Synthesis report.

The Verilog-2001, SystemVerilog, and VHDL language definitions provide specific syntax for specifying attributes, but in Verilog-1995 HDL, you must embed attribute assignments in comments. You can enter attributes in your code using the syntax in Examples 8–18, 8–19, and 8–20, where <*attribute*>, <*attribute type*>, <*value*>, <*object*>, and <*object type*> are variables, and the entry in brackets is optional. The examples in this chapter demonstrate each syntax form.

Verilog HDL is case-sensitive; therefore, synthesis attributes are also case-sensitive.

Example 8–18. Synthesis Attributes in Verilog-1995 HDL

```
// synthesis <attribute> [ = <value> ]
                    or
/* synthesis <attribute> [ = <value> ] */
```

Verilog-1995 comment-embedded attributes, as shown in Example 8–18, must be used as a suffix to (that is, placed after) the declaration of an item and must appear before the semicolon when one is required.



You cannot use the open one-line comment in Verilog HDL when a semicolon is required at the end of the line, because it is not clear to which HDL element the attribute applies. For example, you cannot make an attribute assignment such as `reg r; // synthesis` <*attribute*> because the attribute could be read as part of the next line.

To apply multiple attributes to the same instance in Verilog-1995, separate the attributes with spaces, as follows:

`//synthesis` <*attribute1*> [= <*value*>] <*attribute2*> [= <*value*>]

For example, to set the `maxfan` attribute to `16` (Refer to "Maximum Fan-Out" on page 8–50 for details) and set the `preserve` attribute (refer to "Preserve Registers" on page 8–44 for details) on a register called `my_reg`, use the following syntax:

```
reg my_reg /* synthesis maxfan = 16 preserve */;
```

In addition to the `synthesis` keyword as shown above, the keywords `pragma`, `synopsys`, and `exemplar` are supported for compatibility with other synthesis tools. The keyword `altera` is also supported, which allows you to add synthesis attributes that will be recognized only by Quartus II integrated synthesis and not by other tools that recognize the same synthesis attribute.



Because formal verification tools do not recognize the `exemplar`, `pragma`, and `altera` keywords, avoid using these attribute keywords when using formal verification.

Example 8–19. Synthesis Attributes in Verilog-2001 and SystemVerilog

```
(* <attribute> [ = <value> ] *)
```

Verilog-2001 attributes, as shown in Example 8–19, must be used as a prefix to (that is, placed before) a declaration, module item, statement, or port connection, and used as a suffix to (that is, placed after) an operator or a Verilog HDL function name in an expression.



Because formal verification tools do not recognize the syntax, the Verilog-2001 attribute syntax is not supported when using formal verification.

To apply multiple attributes to the same instance in Verilog-2001 or SystemVerilog, separate the attributes with commas, as shown in the following example:

(* <*attribute1*> [= <*value1*>], <*attribute2*> [= <*value2*>] *)

For example, to set the `maxfan` attribute to `16` (refer to"Maximum Fan-Out" on page 8–50 for details) and set the `preserve` attribute (refer to "Preserve Registers" on page 8–44 for details) on a register called `my_reg`, use the following syntax:

```
(* preserve, maxfan = 16 *) reg my_reg;
```

Example 8–20. Synthesis Attributes in VHDL

```
attribute <attribute> : <attribute type> ;
attribute <attribute> of <object> : <object type> is <value>;
```

VHDL attributes, as shown in Example 8–20, declare the attribute type and then apply it to a specific object. For VHDL designs, all supported synthesis attributes are declared in the `altera_syn_attributes` package in the Altera library. You can call this library from your VHDL code to declare the synthesis attributes, as follows:

```
LIBRARY altera;
USE altera.altera_syn_attributes.all;
```

Synthesis Directives

The Quartus II software supports synthesis directives, also commonly called compiler directives or pragmas. You can include synthesis directives in Verilog HDL or VHDL code as comments. These directives are not standard Verilog HDL or VHDL commands; synthesis tools use directives to control the synthesis process in a particular manner. Directives do not apply to a specific design node but change the behavior of the synthesis tool from the point where they occur in the HDL source code. Other tools, such as simulators, ignore these directives and treat them as comments.

You can enter synthesis directives in your code using the syntax shown in Example 8–21 and 8–22, where <*directive*> and <*value*> are variables, and the entry in brackets is optional. Notice that for synthesis directives there is no = sign before the value; this is different than the syntax for synthesis attributes. The examples in this chapter demonstrate each syntax form.



Verilog HDL is case-sensitive; therefore, all synthesis directives are also case-sensitive.

Example 8–21. Synthesis Directives in Verilog HDL

```
// synthesis <directive> [ <value> ]
                    or
/* synthesis <directive> [ <value> ] */
```

Example 8–22. Synthesis Directives in VHDL

```
-- synthesis <directive> [ <value> ]
```

In addition to the `synthesis` keyword shown above, the `pragma`, `synopsys`, and `exemplar` keywords are supported in both Verilog HDL and VHDL for compatibility with other synthesis tools. The keyword `altera` is also supported, which allows you to add synthesis directives that will be recognized only by Quartus II integrated synthesis and not by other tools that recognize the same synthesis directive.



Because formal verification tools ignore keywords `exemplar`, `pragma`, and `altera`, avoid using these directive keywords when you are using formal verification to prevent mismatches with the Quartus II results.

Optimization Technique

The **Optimization Technique** logic option specifies the goal for logic optimization during compilation; that is, whether to attempt to achieve maximum speed performance or minimum area usage, or a balance between the two. Table 8–2 lists the settings for this logic option, which you can apply only to a design entity. You can also set this logic option for your whole project on the **Analysis & Synthesis Settings** page in the **Settings** dialog box.

Table 8–2. Optimization Technique Settings

Setting	Description
Area	The Compiler makes the design as small as possible to minimize resource usage.
Speed	The Compiler chooses a design implementation that has the fastest f_{MAX}.
Balanced *(1)*	The Compiler maps part of the design for area and part for speed, providing better area utilization than optimizing for speed, with only a slightly slower f_{MAX} than optimizing for speed.

Note to Table 8–2:
(1) The balanced optimization technique is not supported for all device families.

The default setting varies by device family, and is generally optimized for the best area/speed trade-off. Results are design-dependent and vary depending on which device family you use.

Speed Optimization Technique for Clock Domains

The **Speed Optimization Technique for Clock Domains** logic option specifies that all combinational logic in or between the specified clock domain(s) is optimized for speed.

When this option is set on a particular clock signal, all the logic in this clock domain is optimized for speed during synthesis. The remainder of the design in other clock domains is synthesized with the project-wide **Optimization Technique** that is set in the **Analysis & Synthesis Settings**. The option can also be set from one clock to another clock signal, in which case the logic in paths from registers in the first clock domain to registers in the second clock domain are synthesized for speed. The advantage of using this option over the project-wide setting to optimize for speed is that there is less penalty to the area of the design because a smaller part of the circuit is optimized for speed. This may also have a positive effect on clock speed. This option also has an advantage over setting the **Optimization Technique** on a design entity because that option forces the hierarchical blocks to be synthesized separately. Doing so may increase area and decrease performance due to the lack of optimizations across hierarchies. The **Speed Optimization Technique for Clock**

Domains option does not treat hierarchical entities separately, and can optimize across hierarchical boundaries for logic within the same clock domain.

This option is useful if you have one or more clock domains that do not meet your timing requirements. When there are failing paths within a clock domain, the option can be set on the clock of that clock domain. When there are failing paths between clock domains, the option can be set from one clock domain to the other clock domain.

This option is available for the following device families: Arria™ GX, Stratix® series, Cyclone® series, HardCopy® II, HardCopy Stratix, and MAX® II.

PowerPlay Power Optimization

This logic option controls the power-driven compilation setting of Analysis and Synthesis and determines how aggressively Analysis and Synthesis optimizes the design for power. On the Assignments menu, click **Settings**, and under **Category**, click **Analysis & Synthesis Settings**. This displays the **Analysis & Synthesis Settings** page. The following three settings are available for the **PowerPlay Power Optimization** option:

- **Off**—Analysis and Synthesis does not perform any power optimizations.
- **Normal Compilation**—Analysis and Synthesis performs power optimizations, without reducing design performance.
- **Extra Effort**—Analysis and Synthesis performs additional power optimizations, which may reduce design performance.

This logic option is available for the following device families: Arria GX, Stratix series, Cyclone series, HardCopy II, and MAX II.



For more information about optimizing your design for power utilization, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*. For information about analyzing your power results, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Restructure Multiplexers

This option specifies whether the Quartus II software should extract and optimize buses of multiplexers during synthesis.

This option is useful if your design contains buses of fragmented multiplexers. This option restructures multiplexers more efficiently for area, allowing the design to implement multiplexers with a reduced number of LEs or ALMs. This option is available for the following device families: Arria GX, Stratix series, Cyclone series, HardCopy II, and MAX II.

The **Restructure Multiplexers** option works on entire trees of multiplexers. Multiplexers may arise in different parts of the design through Verilog HDL or VHDL constructs such as the "`if`," "`case`," or "`?:`" statements. When multiplexers from one part of the design feed multiplexers in another part of the design, trees of multiplexers are formed. Multiplexer buses occur most often as a result of multiplexing together vectors in Verilog HDL, or `STD_LOGIC_VECTOR` signals in VHDL. The **Restructure Multiplexers** option identifies buses of multiplexer trees that have a similar structure. When it is turned on, the **Restructure Multiplexers** option optimizes the structure of each multiplexer bus for the target device to reduce the overall amount of logic used in the design.

Results of the multiplexer optimizations are design dependent, but area reductions as high as 20% are possible. The option may negatively affect your design's f_{MAX}.

Table 8–3 lists the settings for the logic option, which you can apply only to a design entity. You can also specify this option on the **Analysis & Synthesis Settings** page in the **Settings** dialog box for your whole project.

Table 8–3. Restructure Multiplexers Settings

Setting	Description
On	Enables multiplexer restructuring to minimize your design area. This setting may reduce the f_{MAX}.
Off	Disables multiplexer restructuring to avoid possible reductions in f_{MAX}.
Auto (Default)	Allows the Compiler to determine whether to enable the option based on your other Quartus II synthesis settings. The option is On when the **Optimization Technique** option is set to **Area** or **Balanced**, and Off when the **Optimization Technique** option is **Speed**. (Note that since the default Optimization Technique is Balanced for many device families, including the Stratix series, this option is turned on by default for those families.)

After you have compiled your design, you can view multiplexer restructuring information in the **Multiplexer Restructuring Statistics** report in the **Multiplexer Statistics** folder under **Analysis & Synthesis Optimization Results** in the **Analysis & Synthesis** section of the **Compilation Report**. Table 8–4 describes the information that is listed in the **Multiplexer Restructuring Statistics** report table for each bus of multiplexers.

Table 8–4. Multiplexer Information in the Multiplexer Restructuring Statistics Report

Heading	Description
Multiplexer Inputs	The number of different choices that are multiplexed together.
Bus Width	The width of the bus in bits.
Baseline Area	An estimate of how many logic cells are needed to implement the bus of multiplexers (before any multiplexer restructuring takes place). This estimate can be used to identify any large multiplexers in the design.
Area if Restructured	An estimate of how many logic cells are needed to implement the bus of multiplexers if **Multiplexer Restructuring** is applied.
Saving if Restructured	An estimate of how many logic cells are saved if **Multiplexer Restructuring** is applied.
Registered	An indication of whether registers are present on the multiplexer outputs. **Multiplexer Restructuring** uses the secondary control signals of a register (such as synchronous clear and synchronous-load) to further reduce the amount of logic needed to implement the bus of multiplexers.
Example Multiplexer Output	The name of one of the multiplexers' outputs. This name can help determine where in the design the multiplexer bus originated.

For more information about optimizing for multiplexers, refer to the *Multiplexers* section of the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

State Machine Processing

This logic option specifies the processing style used to compile a state machine. Table 8–5 lists the settings for this logic option, which you can apply to a state machine name or to a design entity containing a state machine. You can also set this option for your whole project on the **Analysis & Synthesis Settings** page in the **Settings** dialog box.

Table 8–5. State Machine Processing Settings

Setting	Description
Auto (Default)	Allows the Compiler to choose what it determines to be the best encoding for the state machine
Minimal Bits	Uses the least number of bits to encode the state machine
One-Hot	Encodes the state machine in the one-hot style. See the example below for details.
User-Encoded	Encodes the state machine in the manner specified by the user
Sequential	Uses a binary encoding in which the first enumeration literal in the Enumeration Type has encoding `0`, the second `1`, and so on.
Gray	Uses an encoding in which the encodings for adjacent enumeration literals differ by exactly one bit. An N-bit gray code can represent 2^N values.
Johnson	Uses an encoding similar to a gray code, in which each state only has one bit different from its neighboring states. Each state is generated by shifting the previous state's bits to the right by 1; the most significant bit of each state is the negation of the least significant bit of the previous state. An N-bit Johnson code can represent at most 2N states but requires less logic than a gray encoding.

The default state machine encoding, which is Auto, uses one-hot encoding for FPGA devices and minimal-bits encoding for CPLDs. These settings achieve the best results on average, but another encoding style might be more appropriate for your design, so this option allows you to control the state machine encoding.

For guidelines to ensure that your state machine is inferred and encoded correctly, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

For one-hot encoding, the Quartus II software does not guarantee that each state has one bit set to one and all other bits to zero. Quartus II integrated synthesis creates one-hot register encoding by using standard

one-hot encoding and then inverting the first bit. This results in an initial state with all zero values, and the remaining states have two 1 values. Quartus II integrated synthesis encodes the initial state with all zeros for the state machine power-up because all device registers power up to a low value. This encoding has the same properties as true one-hot encoding: each state can be recognized by the value of one bit. For example, in a one-hot-encoded state machine with five states including an initial or reset state, the software uses the following register encoding:

```
State 0     0 0 0 0 0
State 1     0 0 0 1 1
State 2     0 0 1 0 1
State 3     0 1 0 0 1
State 4     1 0 0 0 1
```

If the **State Machine Processing** logic option is set to **User-Encoded** in a Verilog HDL design, the software starts with the original design values for the state constants. For example, a Verilog HDL design can contain a declaration such as the following:

parameter S0 = 4'b1010, S1 = 4'b0101, ...

If the software infers states `S0, S1, . . .` it uses the encoding `4'b1010, 4'b0101, . . .`. If necessary, the software inverts bits in a user-encoded state machine to ensure that all bits of the reset state of the state machine are zero.

To assign your own state encoding with the **User-Encoded** setting of the **State Machine Processing** option in a VHDL design, you must apply specific binary encoding to the elements of an enumerated type because enumeration literals have no numeric values in VHDL. Use the `syn_encoding` synthesis attribute to apply your encoding values. Refer to "Manually Specifying State Assignments Using the syn_encoding Attribute" for more information.

For information about the **Safe State Machine** option, refer to "Safe State Machines" on page 8–40.

Manually Specifying State Assignments Using the syn_encoding Attribute

The Quartus II software infers state machines from enumerated types and automatically assigns state encoding based on "State Machine Processing" on page 8–34. With this logic option, you can choose the value **User-Encoded** to use the encoding from your HDL code. However,

in standard VHDL code, you cannot specify user encoding in the state machine description because enumeration literals have no numeric values in VHDL.

To assign your own state encoding for the **User-Encoded State Machine Processing** setting, use the `syn_encoding` synthesis attribute to apply specific binary encodings to the elements of an enumerated type or to specify an encoding style. The Quartus II software can implement Enumeration Types with the different encoding styles shown in Table 8–6.

Table 8–6. syn_encoding Attribute Values

Attribute Value	Description
`"default"`	Use an encoding based on the number of enumeration literals in the Enumeration Type. If there are fewer than five literals, use the `"sequential"` encoding. If there are more than five but fewer than 50 literals, use a `"one-hot"` encoding. Otherwise, use a `"gray"` encoding.
`"sequential"`	Use a binary encoding in which the first enumeration literal in the Enumeration Type has encoding 0, the second 1, and so on.
`"gray"`	Use an encoding in which the encodings for adjacent enumeration literals differ by exactly one bit. An N-bit gray code can represent 2^N values.
`"johnson"`	Use an encoding similar to a gray code. An N-bit Johnson code can represent at most 2N states but requires less logic than a gray encoding.
`"one-hot"`	The default encoding style requiring N bits, where N is the number of enumeration literals in the Enumeration Type.
`"compact"`	Use an encoding with the fewest bits.

The `syn_encoding` attribute must follow the enumeration type definition but precede its use.

In Example 8–23, the `syn_encoding` attribute associates a binary encoding with the states in the enumerated type `count_state`. In this example, the states are encoded with the following values: zero = “11”, one = “01”, two = “10”, three = “00”.

Example 8–23. Specifying User Encoded States with the syn_encoding Attribute in VHDL

```
ARCHITECTURE rtl OF my_fsm IS
   TYPE count_state is (zero, one, two, three);
   ATTRIBUTE syn_encoding : STRING;
   ATTRIBUTE syn_encoding OF count_state : TYPE IS "11 01 10 00";
   SIGNAL present_state, next_state : count_state;
BEGIN
```

You can also use the `syn_encoding` attribute in Verilog HDL to direct the synthesis tool to use the encoding from your HDL code, instead of using the **State Machine Processing** option.

The `syn_encoding` value "user" instructs the Quartus II software to encode each state with its corresponding value from the Verilog source code. By changing the values of your state constants, you can change the encoding of your state machine.

Example 8–24. Specifying User Encoded States with the syn_encoding Attribute in Verilog-2001

```
(* syn_encoding = "user" *) reg [1:0] state;
parameter init = 0, last = 3, next = 1, later = 2;
always @ (state) begin
    case (state)
        init:
            out = 2'b01;
        next:
            out = 2'b10;
        later:
            out = 2'b11;
        last:
            out = 2'b00;
    endcase
end
```

In Example 8–24, the states will be encoded as follows:

```
init = "00"
last = "11"
next = "01"
later = "10"
```

Without the `syn_encoding` attribute, the Quartus II software would encode the state machine based on the current value of the **State Machine Processing** logic option.

If you are also specifying a safe state machine (as described in "Safe State Machines" on page 8–40), separate the encoding style value in the quotation marks with the safe value with a comma, as follows: "`safe, one-hot`" or "`safe, gray`".

Manually Specifying Enumerated Types Using the enum_encoding Attribute

By default, the Quartus II software one-hot encodes all user-defined Enumerated Types. With the `enum_encoding` attribute, you can specify the logic encoding for an Enumerated Type and override the default `one-hot` encoding to improve the logic efficiency.



If an Enumerated Type represents the states of a state machine, using the `enum_encoding` attribute to specify a manual state encoding prevents the Compiler from recognizing state machines based on the Enumerated Type. Instead, the Compiler processes these state machines as "regular" logic using the encoding specified by the attribute, and they are not listed as state machines in the **Report** window for the project. If you wish to control the encoding for a recognized state machine, use the **State Machine Processing** logic option and the `syn_encoding` synthesis attribute.

To use the `enum_encoding` attribute in a VHDL design file, associate the attribute with the Enumeration Type whose encoding you want to control. The `enum_encoding` attribute must follow the Enumeration Type Definition but precede its use. In addition, the attribute value must be a string literal that specifies either an arbitrary user encoding or an encoding style of `"default"`,`"sequential"`,`"gray"`,`"johnson"`,or `"one-hot"`.

An arbitrary user encoding consists of a space-delimited list of encodings. The list must contain as many encodings as there are enumeration literals in your Enumeration Type. In addition, the encodings must all have the same length, and each encoding must consist solely of values from the `std_ulogic` type declared by the `std_logic_1164` package in the `IEEE` library. In the code fragment of Example 8–25, the `enum_encoding` attribute specifies an arbitrary user encoding for the Enumeration Type `fruit`.

Example 8–25. Specifying an Arbitrary User Encoding for Enumerated Type

```
type fruit is (apple, orange, pear, mango);
attribute enum_encoding : string;
attribute enum_encoding of fruit : type is "11 01 10 00";
```

In this example, the enumeration literals are encoded as:

```
apple   = "11"
orange  = "01"
pear    = "10"
mango   = "00"
```

You may wish to specify an encoding style, rather than a manual user encoding, especially when the Enumeration Type has a large number of enumeration literals. The Quartus II software can implement Enumeration Types with the different encoding styles shown in Table 8–7.

Table 8–7. enum_encoding Attribute Values

Attribute Value	Description
`"default"`	Use an encoding based on the number of enumeration literals in the Enumeration Type. If there are fewer than five literals, use the `"sequential"` encoding. If there are more than five but fewer than 50 literals, use a `"one-hot"` encoding. Otherwise, use a `"gray"` encoding.
`"sequential"`	Use a binary encoding in which the first enumeration literal in the Enumeration Type has encoding `0`, the second `1`, and so on.
`"gray"`	Use an encoding in which the encodings for adjacent enumeration literals differ by exactly one bit. An N-bit gray code can represent 2^N values.
`"johnson"`	Use an encoding similar to a gray code. An N-bit Johnson code can represent at most 2N states but requires less logic than a gray encoding.
`"one-hot"`	The default encoding style requiring *N* bits, where *N* is the number of enumeration literals in the Enumeration Type.

Observe that in Example 8–25, the `enum_encoding` attribute manually specified a gray encoding for the Enumeration Type `fruit`. This example could be written more concisely by specifying the `"gray"` encoding style instead of a manual encoding, as shown in Example 8–26.

Example 8–26. Specifying the "gray" Encoding Style or Enumeration Type

```
type fruit is (apple, orange, pear, mango);
attribute enum_encoding : string;
attribute enum_encoding of fruit : type is "gray";
```

Safe State Machines

The **Safe State Machine** option and corresponding `syn_encoding` attribute value `safe` specify that the software should insert extra logic to detect an illegal state and force the state machine's transition to the reset state.

It is possible for a finite state machine to enter an illegal state—meaning the state registers contain a value that does not correspond to any defined state. By default, the behavior of the state machine that enters an illegal state is undefined. However, you can set the `syn_encoding` attribute to `safe` or use the **Safe State Machine** logic option if you want the state machine to recover deterministically from an illegal state. Use this option if you have asynchronous inputs to your state machine. The most common cause of this situation is a state machine that has control inputs that come from another clock domain, such as the control logic for a clock-crossing FIFO, because the state machine must have inputs from another clock domain. An alternative is to add synchronizer registers to the inputs.

It is important to note that the `safe` state machine value does not use any user-defined default logic from your HDL code that corresponds to unreachable states. Verilog HDL and VHDL allow you to explicitly specify a behavior for all states in the state machine, including unreachable states. However, synthesis tools detect if state machine logic is unreachable and minimize or remove the logic. Any flag signals or logic used in the design to indicate such an illegal state are also removed. If the state machine is implemented as safe, the recovery logic forces its transition from an illegal state to the reset state.

The **Safe State Machine** option can be set globally, or on individual state machines. To set this option, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Analysis & Synthesis Settings** dialog box appears.
4. In the **Existing option settings** list, select **Safe State Machine**.
5. Under **Option**, in the **Setting** list, select **On**.
6. Click **OK**.

7. Click **OK** to close the **Settings** dialog box.

You can also use the Assignment Editor to turn on the **Safe State Machine** option for specific state machines.

You can set the `syn_encoding safe` attribute on a state machine in HDL, as shown in Example 8–27, 8–28, and 8–29.

Example 8–27. Verilog HDL Example of a Safe State Machine Attribute

```
reg [2:0] my_fsm /* synthesis syn_encoding = "safe" */;
```

Example 8–28. Verilog-2001 Example of a Safe State Machine Attribute

```
(* syn_encoding = "safe" *) reg [2:0] my_fsm;
```

Example 8–29. VHDL Example of a Safe State Machine Attribute

```
ATTRIBUTE syn_encoding OF my_fsm : TYPE IS "safe";
```

If you are also specifying an encoding style (as described in "Manually Specifying State Assignments Using the syn_encoding Attribute" on page 8–35), separate the encoding style value in the quotation marks with the safe value with a comma, as follows: `"safe, one-hot"` or `"safe, gray"`.

Safe state machine implementation can result in a noticeable area increase for the design. Therefore, Altera recommends that you set this option only on the critical state machines in the design where the safe mode is required, such as a state machine that uses inputs from asynchronous clock domains. You can also reduce the necessity of this option by correctly synchronizing inputs coming from other clock domains.

Note that if the `safe` state machine assignment is made on an instance that is not recognized as a state machine, or an entity that contains a state machine, the software takes no action. You must restructure the code so that the instance is recognized and properly inferred as a state machine.



For guidelines to ensure that your state machine is inferred correctly, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Power-Up Level

This logic option causes a register (flipflop) to power up with the specified logic level, either **High** (1) or **Low** (0). Registers in the device core hardware power up to 0 in all Altera devices. For the register to power up with a logic level **High** specified using this option, the Compiler performs an optimization referred to as NOT-gate push back on the register. NOT-gate push back adds an inverter to the input and the output of the register so that the reset and power-up conditions will appear to be high and the device operates as expected. The register itself still powers up low, but the register output is inverted so the signal arriving at all destinations is high. This option is available for all Altera devices supported by the Quartus II software except MAX® 3000A and MAX 7000S devices.

The **Power-Up Level** option supports wildcard characters, and you can apply this option to any register, registered logic cell WYSIWYG primitive, or to a design entity containing registers if you want to set the power level for all registers in the design entity. If this option is assigned to a registered logic cell WYSIWYG primitive, such as an atom primitive from a third-party synthesis tool, you must turn on the **Perform WYSIWYG Primitive Resynthesis** logic option for it to take effect. You can also apply the option to a pin with the logic configurations described in the following list:

- If this option is turned on for an input pin, the option is transferred automatically to the register that is driven by the pin if the following conditions are present:
  - There is no logic, other than inversion, between the pin and the register
  - The input pin drives the data input of the register
  - The input pin does not fan-out to any other logic
- If this option is turned on for an output or bidirectional pin, it is transferred automatically to the register that feeds the pin, if the following conditions are present:
  - There is no logic, other than inversion, between the register and the pin
  - The register does not fan-out to any other logic

Inferred Power-Up Levels

Quartus II integrated synthesis reads default values for registered signals defined in Verilog HDL and VHDL code, and converts the default values into Power-Up Level settings. The software also synthesizes variables that are assigned values in Verilog HDL initial blocks into power-up

conditions. Synthesis of these default and initial constructs enables the design's synthesized behavior to match, as closely as possible, the power-up state of the HDL code during a functional simulation.

For example, the following register declarations all set a power-up level of V_{CC} or a logic value "1":

```
signal q : std_logic = '1';  -- power-up to VCC

reg q = 1'b1;  // power-up to VCC

reg q;
initial begin q = 1'b1; end  // power-up to VCC
```



For more information about `NOT` gate push-back, the power-up states for Altera devices, and how the power-up level is affected by set and reset control signals, refer to *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Power-Up Don't Care

This logic option allows the compiler to optimize registers in the design which do not have a defined power-up condition. This option is turned on by default.

For example, your design may have a register with its D input tied to V_{CC}, and with no clear signal or other secondary signals. If this option is enabled, the compiler can choose for the register to power up to V_{CC}. Therefore, the output of the register is always V_{CC}. The compiler can remove the register and connect its output to V_{CC}. If you turn this option off or if you set a Power-Up Level assignment of low for this register, the register transitions from GND to V_{CC} when the design starts up on the first clock signal. Thus, the register is not stuck at V_{CC} and cannot be removed. Similarly, if the register has a clear signal, it will not be removed because after the clear is asserted, the register will again transition to GND and back to V_{CC}.

If the Compiler performs a Power-Up Don't Care optimization that allows it to remove a register, it issues a message indicating it is doing so.

This project-wide option does not apply to registers that have the **Power-Up Level** logic option set to either **High** or **Low**.

Remove Duplicate Registers

If you turn on this logic option, the Compiler removes registers that are identical to another register. If two registers generate the same logic, the Compiler removes the second one, and the first one fans out to the second one's destinations. Also, if the deleted register has different logic option assignments, the Compiler ignores them. This option is turned on by default.

Typically, you should use this option only if you want to prevent the Compiler from removing duplicate registers. That is, you should use this option only with the **Off** setting. You can apply this option to an individual register or a design entity that contains registers.

Remove Redundant Logic Cells

This logic option removes redundant `LCELL` primitives or WYSIWYG cells. The option is off by default to preserve logic cells that have been used intentionally. If you turn on this option, the Compiler optimizes a circuit for area and speed. You can set this option globally or apply it to individual nodes and entities. If you turn on the option at the global level, you can use the `keep` attribute or **Implement as Output of Logic Cell** logic option to preserve specific wire signals or nodes (refer to "Keep Combinational Node/Implement as Output of Logic Cell" on page 8–47).

Preserve Registers

This attribute and logic option directs the Compiler not to minimize or remove a specified register during synthesis optimizations or register netlist optimizations. Optimizations can eliminate redundant registers and registers with constant drivers; this option prevents a register from being reduced to a constant or merged with a duplicate register. This option can preserve a register so you can observe it during simulation or with the SignalTap II logic analyzer. Additionally, it can preserve registers if you are creating a preliminary version of the design in which secondary signals are not specified. You can also use the attribute to preserve a duplicate of an I/O register so that one copy can be placed in an I/O cell and the second can be placed in the core. By default, the software may remove one of the two duplicate registers. In this case, the `preserve` attribute can be added to both registers to prevent this.



This option cannot preserve registers that have no fan-out. To prevent the removal of registers with no fan-out, refer to "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 8–46.

The **Preserve Registers** option prevents a register from being inferred as a state machine.

You can set the **Preserve Registers** logic option in the Quartus II GUI or you can set the `preserve` attribute in your HDL code, as shown in Example 8–30, 8–31, and 8–32. In these examples, the `my_reg` register is preserved.



In addition to `preserve`, the Quartus II software supports the `syn_preserve` attribute name for compatibility with other synthesis tools.

Example 8–30. Verilog HDL Example of a syn_preserve Attribute

```
reg my_reg /* synthesis syn_preserve = 1 */;
```

Example 8–31. Verilog-2001 Example of a syn_preserve Attribute

```
(* syn_preserve = 1 *) reg my_reg;
```



The `" = 1"` after the `"preserve"` in Example 8–30 and 8–31 is optional, because the assignment uses a default value of `1` when it is specified.

Example 8–32. VHDL Example of a preserve Attribute

```
signal my_reg : stdlogic;
attribute preserve : boolean;
attribute preserve of my_reg : signal is true;
```

Disable Register Merging/Don't Merge Register

This logic option and attribute prevents the specified register from being merged with other registers, and prevents other registers from being merged with the specified register. When applied to a design entity, it applies to all registers in the entity.

You can use this option to instruct the Compiler to correctly use your timing constraints for the register during synthesis. For example, if the register has a multicycle constraint, this option prevents the Compiler from merging other registers into the specified register, avoiding unintended timing effects and functional differences.

This option differs from the **Preserve Register** option because it does not prevent a register with constant drivers or a redundant register from being removed. In addition, this option prevents other registers from merging with the specified register.

You can set the **Disable Register Merging** logic option in the Quartus II GUI, or you can set the `dont_merge` attribute in your HDL code, as shown in Example 8–33, 8–34, and 8–35. In these examples, the `my_reg` register is prevented from merges.

Example 8–33. Verilog HDL Example of a dont_merge Attribute

```
reg my_reg /* synthesis dont_merge */;
```

Example 8–34. Verilog-2001 Example of a dont_merge Attribute

```
(* dont_merge *) reg my_reg;
```

Example 8–35. VHDL Example of a dont_merge Attribute

```
signal my_reg : stdlogic;
attribute dont_merge : boolean;
attribute dont_merge of my_reg : signal is true;
```

Noprune Synthesis Attribute/Preserve Fan-out Free Register Node

This synthesis attribute and corresponding logic option direct the Compiler to preserve a fan-out-free register through the entire compilation flow. This is different from the **Preserve Registers** option, which prevents a register from being reduced to a constant or merged with a duplicate register. Standard synthesis optimizations remove nodes that do not directly or indirectly feed a top-level output pin. This option can retain a register so you can observe it in the Simulator or the SignalTap II logic analyzer. Additionally, it can retain registers if you are creating a preliminary version of the design in which the registers' fan-out logic is not specified. This option is supported for inferred registers in the following device families: Arria GX, Stratix series, Cyclone series, and MAX II.

You can set the **Preserve Fan-out Free Register Node** logic option in the Quartus II GUI, or you can set the `noprune` attribute in your HDL code, as shown in Example 8–36, 8–37, and 8–38. In these examples, the `my_reg` register is preserved.



You must use the noprune attribute instead of the logic option if the register has no immediate fan-out in its module or entity. If you do not use the synthesis attribute, registers with no fan-out are removed (or "pruned") during analysis and elaboration before the logic synthesis stage applies any logic options. If the register has no fan-out in the full design, but has fan-out within its module or entity, you can use the logic option to retain the register through compilation.

The attribute name syn_noprune is supported for compatibility with other synthesis tools.

Example 8–36. Verilog HDL Example of a syn_noprune Attribute

```
reg my_reg /* synthesis syn_noprune */;
```

Example 8–37. Verilog-2001 Example of a noprune Attribute

```
(* noprune *) reg my_reg;
```

Example 8–38. VHDL Example of a noprune Attribute

```
signal my_reg : stdlogic;
attribute noprune: boolean;
attribute noprune of my_reg : signal is true;
```

Keep Combinational Node/Implement as Output of Logic Cell

This synthesis attribute and corresponding logic option direct the Compiler to keep a wire or combinational node through logic synthesis minimizations and netlist optimizations. A wire that has a keep attribute or a node that has the **Implement as Output of Logic Cell** logic option applied becomes the output of a logic cell in the final synthesis netlist, and the name of the logic cell will be the same as the name of the wire or node. You can use this directive to make combinational nodes visible to the SignalTap II logic analyzer.



The option cannot keep nodes that have no fan-out. Node names cannot be maintained for wires with tri-state drivers, or if the signal feeds a top-level pin of the same name (in this case the node name is changed to a name such as <*net name*>~buf0).

You can set the **Implement as Output of Logic Cell** logic option in the Quartus II GUI, or you can set the keep attribute in your HDL code, as shown in Example 8–39, 8–40, and 8–41. In these examples, the Compiler maintains the node name my_wire.



In addition to keep, the Quartus II software supports the syn_keep attribute name for compatibility with other synthesis tools.

Example 8–39. Verilog HDL Example of a keep Attribute

```
wire my_wire /* synthesis keep = 1 */;
```

Example 8–40. Verilog-2001 Example of a keep Attribute

```
(* keep = 1 *) wire my_wire;
```

Example 8–41. VHDL Example of a syn_keep Attribute

```
signal my_wire: bit;
attribute syn_keep: boolean;
attribute syn_keep of my_wire: signal is true;
```

Don't Retime, Disabling Synthesis Netlist Optimizations

This attribute disables synthesis retiming optimizations on the specified register. When applied to a design entity, it applies to all registers in the entity.

You can use this option to turn off retiming optimizations and prevent node name changes so that the Compiler can correctly use your timing constraints for the register.

You can set the **Netlist Optimizations** logic option to **Never Allow** in the Quartus II GUI to disable retiming along with other synthesis netlist optimizations, or you can set the dont_retime attribute in your HDL code, as shown in Example 8–42 and 8–43. In these examples, the my_reg register is prevented from being retimed.

Example 8–42. Verilog HDL Example of a dont_retime Attribute

```
reg my_reg /* synthesis dont_retime */;
```

Example 8–43. Verilog-2001 Example of a dont_retime Attribute

```
(* dont_retime *) reg my_reg;
```

Example 8–44. VHDL Example of a dont_retime Attribute

```
signal my_reg : std_logic;
attribute dont_retime : boolean;
attribute dont_retime of my_reg : signal is true;
```



For compatibility with third-party synthesis tools, Quartus II integrated synthesis also supports the attribute `syn_allow_retiming`. To disable retiming, set `syn_allow_retiming` to `0` (Verilog) or `false` (VHDL). This attribute does not have any effect when set to `1` or `true`.

Don't Replicate, Disabling Synthesis Netlist Optimizations

This attribute disables synthesis replication optimizations on the specified register. When applied to a design entity, it applies to all registers in the entity.

You can use this option to turn off register replication (or duplication) optimizations so that the Compiler can use your timing constraints for the register.

You can set the **Netlist Optimizations** logic option to **Never Allow** in the Quartus II GUI to disable replication along with other synthesis netlist optimizations, or you can set the `dont_replicate` attribute in your HDL code, as shown in Example 8–45 and 8–46. In these examples, the `my_reg` register is prevented from being replicated.

Example 8–45. Verilog HDL Example of a dont_replicate Attribute

```
reg my_reg /* synthesis dont_replicate  */;
```

Example 8–46. Verilog-2001 Example of a dont_replicate Attribute

```
(* dont_replicate *) reg my_reg;
```

Example 8–47. VHDL Example of a dont_replicate Attribute

```
signal my_reg : std_logic;
attribute dont_replicate : boolean;
attribute dont_replicate of my_reg : signal is true;
```



For compatibility with third-party synthesis tools, Quartus II integrated synthesis also supports the attribute `syn_replicate`. To disable replication, set `syn_replicate` to `0` (Verilog) or `false` (VHDL). This attribute does not have any effect when set to `1` or `true`.

Maximum Fan-Out

This attribute and logic option directs the Compiler to control the number of destinations fed by a node. The Compiler duplicates a node and splits its fan-out until the individual fan-out of each copy falls below the maximum fan-out restriction. You can apply this option to a register or a logic cell buffer, or to a design entity that contains these elements. You can use this option to reduce the load of critical signals, which can improve performance. You can use the option to instruct the Compiler to duplicate (or replicate) a register that feeds nodes in different locations on the target device. Duplicating the register may allow the Fitter to place these new registers closer to their destination logic, minimizing routing delay.

This option is available for all devices supported in the Quartus II software except MAX 3000, MAX 7000, FLEX 10K®, ACEX® 1K, and Mercury™ devices. To turn off the option for a given node if the option is set at a higher level of the design hierarchy, in the **Netlist Optimizations** logic option, select **Never Allow**. If not disabled by the **Netlist Optimizations** option, the maximum fan-out constraint is honored as long as the following conditions are met:

- The node is not part of a cascade, carry, or register cascade chain
- The node does not feed itself
- The node feeds other logic cells, DSP blocks, RAM blocks, and/or pins through data, address, clock enable, etc, but not through any asynchronous control ports (such as asynchronous clear)

The software does not create duplicate nodes in these cases either because there is no clear way to duplicate the node, or, to avoid the possible situation that small differences in timing could produce functional differences in the implementation (in the third condition above where asynchronous control signals are involved). If the constraint cannot be applied because one of these conditions is not met, the Quartus II software issues a message indicating that it ignored maximum fan-out assignment. To instruct the software not to check the node's destinations for possible problems like the third condition, you can set the **Netlist Optimizations** logic option to **Always Allow** for a given node.

If you have enabled any of the Quartus II netlist optimizations that affect registers, add the `preserve` attribute to any registers to which you have set a `maxfan` attribute. The `preserve` attribute ensures that the registers are not affected by any of the netlist optimization algorithms, such as register retiming.

For details about netlist optimizations, refer to the *Netlist Optimization and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

You can set the **Maximum Fan-Out** logic option in the Quartus II GUI, and this option supports wildcard characters. You can also set the `maxfan` attribute in your HDL code, as shown in Example 8–48, 8–49, and 8–50. In these examples, the Compiler duplicates the `clk_gen` register, so its fan-out is not greater than 50.

In addition to `maxfan`, the Quartus II software supports the `syn_maxfan` attribute name for compatibility with other synthesis tools.

Example 8–48. Verilog HDL Example of a syn_maxfan Attribute

```
reg clk_gen /* synthesis syn_maxfan = 50 */;
```

Example 8–49. Verilog-2001 Example of a maxfan Attribute

```
(* maxfan = 50 *) reg clk_gen;
```

Example 8–50. VHDL Example of a maxfan Attribute

```
signal clk_gen : stdlogic;
attribute maxfan : signal ;
attribute maxfan of clk_gen : signal is 50;
```

Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable

The **Auto Clock Enable Replacement** logic option allows the software to find logic that feeds a register and move the logic to the register's clock enable input port. The option is on by default. You can set this option to **Off** for individual registers or design entities to solve fitting or performance issues with designs that have many clock enables. Turning the option off prevents the software from using the register's clock enable port, and the software implements the clock enable functionality using multiplexers in logic cells.

If specific logic is not automatically moved to a clock enable input with the **Auto Clock Enable Replacement** logic option, you can instruct the software to use a direct clock enable signal. Applying the `direct_enable` attribute to a specific signal instructs the software to use the clock enable port of a register to implement the signal. The attribute ensures that the clock enable port is driven directly by the signal, and the signal is not optimized or combined with any other logic.

Example 8–51, 8–52, and 8–53 show how to set this attribute to ensure that the signal is preserved and used directly as a clock enable.



In addition to `direct_enable`, the Quartus II software supports the `syn_direct_enable` attribute name for compatibility with other synthesis tools.

Example 8–51. Verilog HDL Example of a direct_enable attribute

```
wire my_enable /* synthesis direct_enable = 1 */ ;
```

Example 8–52. Verilog-2001 Example of a syn_direct_enable attribute

```
(* syn_direct_enable *) wire my_enable;
```

Example 8–53. VHDL Example of a direct_enable attribute

```
attribute direct_enable: boolean;
attribute direct_enable of my_enable: signal is true;
```

Megafunction Inference Control

The Quartus II Compiler automatically recognizes certain types of HDL code and infers the appropriate megafunction. The software uses the Altera megafunction code when compiling your design, even when you do not specifically instantiate the megafunction. The software infers megafunctions to take advantage of logic that is optimized for Altera devices. The area and performance of such logic may be better than the results obtained by inferring generic logic from the same HDL code.

Additionally, you must use megafunctions to access certain architecture-specific features, such as RAM, digital signal processing (DSP) blocks, and shift registers, that generally provide improved performance compared with basic logic cells.



For details about coding style recommendations when targeting megafunctions in Altera devices, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

The Quartus II software provides options to control the inference of certain types of megafunctions, as described in the following subsections.

Multiply-Accumulators and Multiply-Adders

Use the **Auto DSP Block Replacement** logic option to control DSP block inference for multiply-accumulations and multiply-adders. This option is turned on by default. To disable inference, turn off this option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box, or disable the option for a specific block with the **Assignment Editor**.



Any registers that the software maps to the altmult_accum and altmult_add megafunctions and places in DSP blocks are not available in the Simulator because their node names do not exist after synthesis.

Shift Registers

Use the **Auto Shift Register Replacement** logic option to control shift register inference. This option is turned on by default. To disable inference, turn off this option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box, or for a specific block with the **Assignment Editor**. The software may not infer small shift registers because small shift registers typically do not benefit from implementation in dedicated memory. However, you can use the **Allow Any Shift Register Size for Recognition** logic option to instruct synthesis to infer a shift register even when its size is considered too small.



The registers that the software maps to the altshift_taps megafunction and places in RAM are not available in the Simulator because their node names do not exist after synthesis.

The **Auto Shift Register Replacement** logic option is turned off automatically when a formal verification tool is selected in the **EDA Tool Settings**. The software issues a warning and lists shift registers that would have been inferred if no formal verification tool was selected in the compilation report. To allow the use of a megafunction for the shift register in the formal verification flow, you can either instantiate a shift register explicitly using the MegaWizard® Plug-in Manager or black box the shift register in a separate entity/module.

RAM and ROM

Use the **Auto RAM Replacement** and **Auto ROM Replacement** logic options to control RAM and ROM inference, respectively. These options are turned on by default. To disable inference, turn off the appropriate option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box, or disable the option for a specific block with the **Assignment Editor**.



Although inferred shift registers are implemented in RAM blocks, you cannot turn off the Auto RAM replacement option to disable shift register replacement. Use the **Auto Shift Register Replacement** option (refer to "Shift Registers").

The software may not infer very small RAM or ROM blocks because very small memory blocks can typically be implemented more efficiently by using the registers in the logic. However, you can use the **Allow Any RAM Size for Recognition** and **Allow Any ROM Size for Recognition** logic options to instruct synthesis to infer a memory block even when its size is considered too small.



The **Auto ROM Replacement** logic option is automatically turned off when a formal verification tool is selected in the **EDA Tool Settings** page. A warning is issued and a report panel lists ROMs that would have been inferred if no formal verification tool was selected. To allow the use of a megafunction for the shift register in the formal verification flow, you can either instantiate a ROM explicitly using the MegaWizard Plug-In Manager or create a black box for the ROM in a separate entity/module.

Although formal verification tools do not support inferred RAM blocks, because of the importance of inferring RAM in many designs, the **Auto RAM Replacement** logic option remains on when a formal verification tool is selected in the **EDA Tool Settings** page. The Quartus II software automatically black boxes any module or entity that contains a RAM block that is inferred. The software issues a warning and lists the black box that is created in the compilation report. This block box allows formal verification tools to proceed; however, the entire module or entity containing the RAM cannot be verified in the tool. Altera recommends that you explicitly instantiate RAM blocks in separate modules or entities so that as much logic as possible can be verified by the formal verification tool.

RAM to Logic Cell Conversion

The **Auto RAM to Logic Cell Conversion** option allows the Quartus II integrated synthesis to convert RAM blocks that are small in size to logic cells if the logic cell implementation is deemed to give better quality of results. Only single-port or simple-dual port RAMs with no initialization files can be converted to logic cells. This option is off by default. You can set this option globally or apply it to individual RAM nodes.

For FLEX 10K, APEX, Arria GX, and the Stratix series of devices, the software uses the following rules to determine whether a RAM should be placed in logic cells or a dedicated RAM block:

- If the number of words is less than 16, use a RAM block if the total number of bits is greater than or equal to 64.
- If the number of words is greater than or equal to 16, use a RAM block if the total number of bits is greater than or equal to 32.
- Otherwise, implement the RAM in logic cells.

For the Cyclone series of devices, the software uses the following rules:

- If the number of words is greater than or equal to 64, use a RAM block.
- If the number of words is greater than or equal to 16 and less than 64, use a RAM block if the total number of bits is greater than or equal to 128.
- Otherwise, implement the RAM in logic cells.

RAM Style and ROM Style—for Inferred Memory

These attributes specify the implementation for an inferred RAM or ROM block. You can specify the type of TriMatrix™ embedded memory block to be used, or specify the use of standard logic cells (LEs or ALMs). The attributes are supported only for device families with TriMatrix embedded memory blocks.

The `ramstyle` and `romstyle` attributes take a single string value. The values `"M512"`, `"M4K"`, `"M-RAM"`, `"MLAB"`, `"M9K"`, and `"M144K"` (as applicable for the target device family) indicate the type of memory block to use for the inferred RAM or ROM. If you set the attribute to a block type that does not exist in the target device family, the software generates a warning and ignores the assignment. The value `logic` indicates that the RAM or ROM should be implemented in regular logic rather than dedicated memory blocks. You can set the attribute on a module or entity, in which case it specifies the default implementation style for all inferred memory blocks in the immediate hierarchy. You can also set the attribute

on a specific signal (VHDL) or variable (Verilog HDL) declaration, in which case it specifies the preferred implementation style for that specific memory, overriding the default implementation style.



If you specify a value of `logic`, the memory still appears as a RAM or ROM block in the RTL Viewer, but it is converted to regular logic during a later synthesis step.

In addition to `ramstyle` and `romstyle`, the Quartus II software supports the `syn_ramstyle` attribute name for compatibility with other synthesis tools.

Example 8–54, 8–55, and 8–56 specify that all memory in the module or entity `my_memory_blocks` should be implemented using a specific type of block.

Example 8–54. Verilog-1995 Example of Applying a romstyle Attribute to a Module Declaration

```
module my_memory_blocks (...) /* synthesis romstyle = "M4K" */
```

Example 8–55. Verilog-2001 Example of Applying a ramstyle Attribute to a Module Declaration

```
 (* ramstyle = "M512" *) module my_memory_blocks (...);
```

Example 8–56. VHDL Example of Applying a romstyle Attribute to an Architecture

```
architecture rtl of my_ my_memory_blocks is
attribute romstyle : string;
attribute romstyle of rtl : architecture is "M-RAM";
begin
```

Example 8–57, 8–58, and 8–59 specify that the inferred memory `my_ram` or `my_rom` should be implemented using regular logic instead of a TriMatrix memory block.

Example 8–57. Verilog-1995 Example of Applying a syn_ramstyle Attribute to a Variable Declaration

```
reg [0:7] my_ram[0:63] /* synthesis syn_ramstyle = "logic" */;
```

Example 8–58. Verilog-2001 Example of Applying a romstyle Attribute to a Variable Declaration

```
(* romstyle = "logic" *) reg [0:7] my_rom[0:63];
```

Example 8–59. VHDL Example of Applying a ramstyle Attribute to a Signal Declaration

```
type memory_t is array (0 to 63) of std_logic_vector (0 to 7);
signal my_ram : memory_t;
attribute ramstyle : string;
attribute ramstyle of my_ram : signal is "logic";
```

Turning Off Add Pass-Through Logic to Inferred RAMs/ no_rw_check Attribute Setting

Setting the `no_rw_check` value for the `ramstyle` attribute, or turning off the corresponding global logic option **Add Pass-Through Logic to Inferred RAMs** indicates that your design does not depend on the behavior of the inferred RAM when there are reads and writes to the same address in the same clock cycle. If you specify the attribute or turn off the logic option, the Quartus II software can choose a read-during-write behavior instead of using the read-during-write behavior of your HDL source code.

In some cases, an inferred RAM must be mapped into regular logic cells because it has a read-during-write behavior that is not supported by the TriMatrix memory blocks in your target device. In other cases, the Quartus II software must insert extra logic to mimic read-during-write behavior of the HDL source, increasing the area of your design and potentially reducing its performance. In these cases, you can use the attribute to specify that the software can implement the RAM directly in a TriMatrix memory block without using logic. You can also use the attribute to prevent a warning message for dual-clock RAMs in the case that the inferred behavior in the device does not exactly match the read-during-write conditions described in the HDL code.



For more information about recommended styles for inferring RAM and some of the issues involved with different read-during-write conditions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

To set the **Add Pass-Through Logic to Inferred RAMs** logic option through the Quartus II GUI, click **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Example 8–60 and 8–61 use two addresses and normally require extra logic after the RAM to ensure that the read-during-write conditions in the device match the HDL code. If you don't require a defined read-during-write condition in your design, this extra logic is not required. With the `no_rw_check` attribute, Quartus II integrated synthesis won't generate the extra logic.

Example 8–60. Verilog HDL Inferred RAM Using no_rw_check Attribute

```
module ram_infer (q, wa, ra, d, we, clk);
    output [7:0] q;
    input [7:0] d;
    input [6:0] wa;
    input [6:0] ra;
    input we, clk;
    reg [6:0] read_add;
    (* ramstyle = "no_rw_check" *) reg [7:0] mem [127:0];
    always @ (posedge clk) begin
        if (we)
            mem[wa] <= d;
        read_add <= ra;
    end
    assign q = mem[read_add];
endmodule
```

Example 8–61. VHDL Inferred RAM Using no_rw_check Attribute

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ram IS
    PORT (
        clock: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END ram;

ARCHITECTURE rtl OF ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ram_block: MEM;
    ATTRIBUTE ramstyle : string;
    ATTRIBUTE ramstyle of ram_block : signal is "no_rw_check";
    SIGNAL read_address_reg: INTEGER RANGE 0 to 31;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
            read_address_reg <= read_address;
        END IF;
    END PROCESS;
    q <= ram_block(read_address_reg);
END rtl;
```

RAM Initialization File—for Inferred Memory

The `ram_init_file` attribute specifies the initial contents of an inferred memory in the form of a Memory Initialization File (**.mif**). The attribute takes a string value containing the name of the RAM initialization file.

Example 8–62. Verilog-1995 Example of Applying a ram_init_file Attribute

```
reg [7:0] mem[0:255] /* synthesis ram_init_file
= " my_init_file.mif" */;
```

Example 8–63. Verilog-2001 Example of Applying a ram_init_file Attribute

```
(* ram_init_file = "my_init_file.mif" *) reg [7:0] mem[0:255];
```

Example 8–64. VHDL Example of Applying a ram_init_file Attribute

```
type mem_t is array(0 to 255) of unsigned(7 downto 0);
signal ram : mem_t;
attribute ram_init_file : string;
attribute ram_init_file of ram :
signal is "my_init_file.mif";
```



In VHDL, you can also initialize the contents of an inferred memory by specifying a default value for the corresponding signal. In Verilog HDL, you can use an initial block to specify the memory contents. Quartus II integrated synthesis automatically converts the default value into a MIF for the inferred RAM.

Multiplier Style—for Inferred Multipliers

The `multstyle` attribute specifies the implementation style for multiplication operations (`*`) in your HDL source code. You can use this attribute to specify whether you prefer the Compiler to implement a multiplication operation in general logic or dedicated hardware, if available in the target device.

The `multstyle` attribute takes a string value of `"logic"` or `"dsp"`, indicating a preferred implementation in logic or in dedicated hardware, respectively. In Verilog HDL, apply the attribute to a module declaration, a variable declaration, or a specific binary expression containing the `*` operator. In VHDL, apply the synthesis attribute to a signal, variable, entity, or architecture.



Specifying a `multstyle` of `"dsp"` does not guarantee that the Quartus II software can implement a multiplication in dedicated DSP hardware. The final implementation depends on several things, including the availability of dedicated hardware in the target device, the size of the operands, and whether or not one or both operands are constant.

In addition to `multstyle`, the Quartus II software supports the `syn_multstyle` attribute name for compatibility with other synthesis tools.

When applied to a Verilog HDL module declaration, the attribute specifies the default implementation style for all instances of the `*` operator in the module. For example, in the following code examples, the `multstyle` attribute directs the Quartus II software to implement all multiplications inside module `my_module` in dedicated multiplication hardware.

Example 8–65. Verilog-1995 Example of Applying a multstyle Attribute to a Module Declaration

```
module my_module (...) /* synthesis multstyle = "dsp" */;
```

Example 8–66. Verilog-2001 Example of Applying a multstyle Attribute to a Module Declaration

```
(* multstyle = "dsp" *) module my_module(...);
```

When applied to a Verilog HDL variable declaration, the attribute specifies the implementation style to be used for a multiplication operator whose result is directly assigned to the variable. It overrides the `multstyle` attribute associated with the enclosing module, if present. In Example 8–67 and 8–68, the `multstyle` attribute applied to variable `result` directs the Quartus II software to implement `a * b` in general logic rather than dedicated hardware.

Example 8–67. Verilog-2001 Example of Applying a multstyle Attribute to a Variable Declaration

```
wire [8:0] a, b;
(* multstyle = "logic" *) wire [17:0] result;
assign result = a * b;  //Multiplication must be
                        //directly assigned to result
```

Example 8–68. Verilog-1995 Example of Applying a multstyle Attribute to a Variable Declaration

```
wire [8:0] a, b;
wire [17:0] result /* synthesis multstyle = "logic" */;
assign result = a * b;  //Multiplication must be
                        //directly assigned to result
```

When applied directly to a binary expression containing the * operator, the attribute specifies the implementation style for that specific operator alone and overrides any multstyle attribute associated with the target variable or enclosing module. In Example 8–69, the multstyle attribute indicates that a * b should be implemented in dedicated hardware.

Example 8–69. Verilog-2001 Example of Applying a multstyle Attribute to a Binary Expression

```
wire [8:0] a, b;
wire [17:0] result;
assign result = a * (* multstyle = "dsp" *) b;
```



You cannot use Verilog-1995 attribute syntax to apply the multstyle attribute to a binary expression.

When applied to a VHDL entity or architecture, the attribute specifies the default implementation style for all instances of the * operator in the entity or architecture. In Example 8–70, the multstyle attribute directs the Quartus II software to use dedicated hardware, if possible, for all multiplications inside architecture rtl of entity my_entity.

Example 8–70. VHDL Example of Applying a multstyle Attribute to an Architecture

```
architecture rtl of my_entity is
      attribute multstyle : string;
      attribute multstyle of rtl : architecture is "dsp";
begin
```

When applied to a VHDL signal or variable, the attribute specifies the implementation style to be used for all instances of the * operator whose result is directly assigned to the signal or variable. It overrides the multstyle attribute associated with the enclosing entity or architecture, if present. In Example 8–71, the multstyle attribute associated with signal result directs the Quartus II software to implement a * b in general logic rather than dedicated hardware.

Example 8–71. VHDL Example of Applying a multstyle Attribute to a Signal or Variable

```
signal a, b : unsigned(8 downto 0);
signal result : unsigned(17 downto 0);

attribute multstyle : string;
attribute multstyle of result : signal is "logic";
result <= a * b;
```

Full Case

A Verilog HDL case statement is considered full when its case items cover all possible binary values of the case expression or when a default case statement is present. A `full_case` attribute attached to a case statement header that is not full forces the unspecified states to be treated as a "don't care" value. VHDL case statements must be full, so the attribute does not apply to VHDL.



Using this attribute on a case statement that is not full avoids the latch inference problems discussed in the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.



Latches have limited support in formal verification tools. It is important to ensure that you do not infer latches unintentionally, for example, through an incomplete case statement when using formal verification. Formal verification tools do support the `full_case` synthesis attribute (with limited support for attribute syntax, as described in "Synthesis Attributes" on page 8–26).

When you are using the `full_case` attribute, there is a potential cause for a simulation mismatch between Verilog HDL functional and post-Quartus II simulation because unknown case statement cases may still function like latches during functional simulation. For example, a simulation mismatch may occur with the code in Example 8–72 when `sel` is `2'b11` because a functional HDL simulation output behaves like a latch while the Quartus II simulation output behaves like "don't care."



Altera recommends making the case statement "full" in your regular HDL code, instead of using the `full_case` attribute.

The case statement in Example 8–72 is not full because not all binary values for `sel` are specified. Because the `full_case` attribute is used, synthesis treats the output as "don't care" when the `sel` input is `2'b11`.

Example 8–72. Verilog HDL Example of a full_case Attribute

```
module full_case (a, sel, y);
    input [3:0] a;
    input [1:0] sel;
    output y;
    reg y;
    always @ (a or sel)
    case (sel) // synthesis full_case
        2'b00: y=a[0];
        2'b01: y=a[1];
        2'b10: y=a[2];
        endcase
endmodule
```

Verilog-2001 syntax also accepts the statements in Example 8–73 in the `case` header instead of the comment form shown in Example 8–72.

Example 8–73. Verilog-2001 Syntax for the full_case Attribute

```
(* full_case *) case (sel)
```

Parallel Case

The `parallel_case` attribute indicates that a Verilog HDL case statement should be considered parallel; that is, only one case item can be matched at a time. Case items in Verilog HDL case statements may overlap. To resolve multiple matching case items, the Verilog HDL language defines a priority relationship among case items in which the case statement always executes the first case item that matches the case expression value. By default, the Quartus II software implements the extra logic required to satisfy this priority relationship.

Attaching a `parallel_case` attribute to a case statement's header allows the Quartus II software to consider its case items as inherently parallel; that is, at most one case item matches the case expression value. Parallel case items reduce the complexity of the generated logic.

In VHDL, the individual choices in a case statement may not overlap, so they are always parallel and this attribute does not apply.

Use this attribute only when the `case` statement is truly parallel. If you use the attribute in any other situation, the generated logic will not match the functional simulation behavior of the Verilog HDL.



Altera recommends that you avoid using the `parallel_case` attribute, due to the possibility of introducing mismatches between Verilog HDL functional and post-Quartus II simulation.

If you specify the supported Verilog HDL version as SystemVerilog-2005 for your design, you can use the SystemVerilog keyword `unique` to achieve the same result as the `parallel_case` directive without causing simulation mismatches.

The following example shows a `casez` statement with overlapping case items. In functional HDL simulation, the three case items have a priority order that depends on the bits in `sel`. For example, `sel[2]` takes priority over `sel[1]`, which takes priority over `sel[0]`. However, the synthesized design may simulate differently because the `parallel_case` attribute eliminates this priority order. If more than one bit of `sel` is high, more than one output (`a`, `b`, or `c`) is high as well, a situation that cannot occur in functional HDL simulation.

Example 8–74. Verilog HDL Example of a parallel_case Attribute

```
module parallel_case (sel, a, b, c);
   input [2:0] sel;
   output a, b, c;
   reg a, b, c;
   always @ (sel)
   begin
      {a, b, c} = 3'b0;
      casez (sel) // synthesis parallel_case
         3'b1??: a = 1'b1;
         3'b?1?: b = 1'b1;
         3'b??1: c = 1'b1;
      endcase
   end
endmodule
```

Verilog-2001 syntax also accepts the statements as shown in Example 8–75 in the `case` (or `casez`) header instead of the comment form, as shown in Example 8–74.

Example 8–75. Verilog-2001 Syntax

```
(* parallel_case *) casez (sel)
```

Translate Off and On / Synthesis Off and On

The `translate_off` and `translate_on` synthesis directives indicate whether the Quartus II software or a third-party synthesis tool should compile a portion of HDL code that is not relevant for synthesis. The `translate_off` directive marks the beginning of code that the synthesis tool should ignore; the `translate_on` directive indicates that synthesis should resume. You can also use the `synthesis_on` and `synthesis_off` directives as a synonym for translate on and off.

A common use of these directives is to indicate a portion of code that is intended for simulation only. The synthesis tool reads synthesis-specific directives and processes them during synthesis; however, third-party simulation tools read the directives as comments and ignore them. Example 8–76 and Example 8–77 show these directives.

Example 8–76. Verilog HDL Example of Translate Off and On

```
// synthesis translate_off
parameter tpd = 2;    // Delay for simulation
#tpd;
// synthesis translate_on
```

Example 8–77. VHDL Example of Translate Off and On

```
-- synthesis translate_off
use std.textio.all;
-- synthesis translate_on
```

If you wish to ignore a portion of code in Quartus II integrated synthesis only, you can use the Altera-specific attribute keyword `altera`. For example, use the `// altera translate_off` and `// altera translate_on` directives to direct Quartus II integrated synthesis to ignore a portion of code that is intended only for other synthesis tools.

Ignore translate_off and synthesis_off Directives

The **Ignore translate_off and synthesis_off directives** logic option directs Quartus II integrated synthesis to ignore the `translate_off` and `synthesis_off` directives described in the previous section. This allows you to compile code that was previously intended to be ignored by third-party synthesis tools, for example, megafunction declarations that were treated as black boxes in other tools but can be compiled in the Quartus II software. To set the **Ignore translate_off and synthesis_off directives** logic option, click **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box.

Read Comments as HDL

The read_comments_as_HDL synthesis directive indicates that the Quartus II software should compile a portion of HDL code that is commented out. This directive allows you to comment out portions of HDL source code that are not relevant for simulation, while instructing the Quartus II software to read and synthesize that same source code. Setting the read_comments_as_HDL directive to on marks the beginning of commented code that the synthesis tool should read; setting the read_comments_as_HDL directive to off indicates the end of the code.



You can use this directive with translate_off and translate_on to create one HDL source file that includes both a megafunction instantiation for synthesis and a behavioral description for simulation.

Because formal verification tools do not recognize the read_comments_as_HDL directive, it is not supported when you are using formal verification.

In Example 8–78 and 8–79, the commented code enclosed by read_comments_as_HDL is visible to the Quartus II Compiler and is synthesized.



Because synthesis directives are case-sensitive in Verilog HDL, you must match the case of the directive, as shown in the following examples.

Example 8–78. Verilog HDL Example of Read Comments as HDL

```
// synthesis read_comments_as_HDL on
// my_rom lpm_rom (.address (address),
//                 .data    (data));
// synthesis read_comments_as_HDL off
```

Example 8–79. VHDL Example of Read Comments as HDL

```
-- synthesis read_comments_as_HDL on
-- my_rom : entity lpm_rom
--   port map (
--    address => address,
--    data    => data,      );
-- synthesis read_comments_as_HDL off
```

Use I/O Flipflops

This attribute directs the Quartus II software to implement input, output, and output enable flipflops (or registers) in I/O cells that have fast, direct connections to an I/O pin, when possible. Applying the `useioff` synthesis attribute can improve I/O performance by minimizing setup, clock-to-output, and clock-to-output enable times. This synthesis attribute is supported using the **Fast Input Register**, **Fast Output Register**, and **Fast Output Enable Register** logic options that can also be set in the **Assignment Editor**.



For more information about which device families support fast input, output, and output enable registers, refer to the device family data sheet, device handbook, or the Quartus II Help.

The `useioff` synthesis attribute takes a Boolean value and can only be applied to the port declarations of a top-level Verilog HDL module or VHDL entity (it is ignored if applied elsewhere). Setting the value to `1` (Verilog HDL) or `TRUE` (VHDL) instructs the Quartus II software to pack registers into I/O cells. Setting the value to 0 (Verilog HDL) or `FALSE` (VHDL) prevents register packing into I/O cells.

In Example 8–80 and 8–81, the `useioff` synthesis attribute directs the Quartus II software to implement the registers `a_reg`, `b_reg`, and `o_reg` in the I/O cells corresponding to the ports `a`, `b`, and `o`, respectively.

Example 8–80. Verilog HDL Example of the useioff Attribute

```
module top_level(clk, a, b, o);
       input clk;
       input [1:0] a, b /* synthesis useioff = 1 */;
       output [2:0] o /* synthesis useioff = 1 */;
       reg [1:0] a_reg, b_reg;
       reg [2:0] o_reg;
       always @ (posedge clk)
       begin
           a_reg <= a;
           b_reg <= b;
           o_reg <= a_reg + b_reg;
       end
       assign o = o_reg;
endmodule
```

Verilog-2001 syntax also accepts the type of statements shown in Example 8–81 and 8–82 instead of the comment form shown in Example 8–80.

Example 8–81. Verilog-2001 Syntax for the useioff Attribute

```
(* useioff = 1 *)   input [1:0] a, b;
(* useioff = 1 *)   output [2:0] o;
```

Example 8–82. VHDL Example of the useioff Attribute

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity useioff_example is
   port (
      clk  : in  std_logic;
      a, b : in  unsigned(1 downto 0);
      o    : out unsigned(1 downto 0));
   attribute useioff : boolean;
   attribute useioff of a : signal is true;
   attribute useioff of b : signal is true;
   attribute useioff of o : signal is true;
end useioff_example;
architecture rtl of useioff_example is
   signal o_reg, a_reg, b_reg : unsigned(1 downto 0);
begin
   process(clk)
   begin
      if (clk = '1' AND clk'event) then
          a_reg <= a;
          b_reg <= b;
          o_reg <= a_reg + b_reg;
      end if;
end process;
   o <= o_reg;
end rtl;
```

Specifying Pin Locations with chip_pin

This attribute enables you to assign pin locations in your HDL source. The attribute can be used only on the ports of the top-level entity or module in the design, and cannot be used to assign pin locations from entities at lower levels of the design hierarchy. You may assign pins only to single-bit or one-dimensional bus ports in your design.

For single-bit ports, the value of the `chip_pin` attribute is the name of the pin on the target device, as specified by the device's pin table.



In addition to chip_pin, the Quartus II software supports the altera_chip_pin_lc attribute name for compatibility with other synthesis tools. When using this attribute in other synthesis tools, some older device families require an "@" symbol in front of each pin assignment. In the Quartus II software, the "@" is optional.

Example 8–83, 8–84, and 8–85 show different ways of assigning input pin my_pin1 to Pin C1 and my_pin2 to Pin 4 on a different target device.

Example 8–83. Verilog-1995 Examples of Applying Chip Pin to a Single Pin

```
input my_pin1 /* synthesis chip_pin = "C1" */;
input my_pin2 /* synthesis altera_chip_pin_lc = "@4" */;
```

Example 8–84. Verilog-2001 Example of Applying Chip Pin to a Single Pin

```
(* chip_pin = "C1" *) input my_pin1;
(* altera_chip_pin_lc = "@4" *) input my_pin2;
```

Example 8–85. VHDL Example of Applying Chip Pin to a Single Pin

```
entity my_entity is
     port(my_pin1: in std_logic; my_pin2: in std_logic;…);
end my_entity;
attribute chip_pin : string;
attribute altera_chip_pin_lc : string;
attribute chip_pin of my_pin1 : signal is "C1";
attribute altera_chip_pin_lc of my_pin2 : signal is "@4"
```

For bus I/O ports, the value of the chip pin attribute is a comma-delimited list of pin assignments. The order in which you declare the port's range determines the mapping of assignments to individual bits in the port. To leave a particular bit unassigned, simply leave its corresponding pin assignment blank.

Example 8–86 assigns my_pin[2] to Pin_4, my_pin[1] to Pin_5, and my_pin[0] to Pin_6.

Example 8–86. Verilog-1995 Example of Applying Chip Pin to a Bus of Pins

```
input [2:0]  my_pin /* synthesis chip_pin = "4, 5, 6" */;
```

Example 8–87 reverses the order of the signals in the bus, assigning my_pin[0] to Pin_4 and my_pin[2] to Pin_6 but leaves my_pin[1] unassigned.

Example 8–87. Verilog-1995 Example of Applying Chip Pin to Part of a Bus

```
input [0:2]  my_pin /* synthesis chip_pin = "4, ,6" */;
```

Example 8–88 assigns `my_pin[2]` to Pin `4` and `my_pin[0]` to Pin `6`, but leaves `my_pin[1]` unassigned.

Example 8–88. VHDL Example of Applying Chip Pin to Part of a Bus of Pins

```
entity my_entity is
     port(my_pin: in std_logic_vector(2 downto 0);…);
end my_entity;

attribute chip_pin of my_pin: signal is "4, , 6";
```

Using altera_attribute to Set Quartus II Logic Options

This attribute enables you to apply Quartus II options and assignments to an object in your HDL source code. You can set this attribute on an entity, architecture, instance, register, RAM block, or I/O pin. You cannot set it on an arbitrary combinational node such as a net. With `altera_attribute`, you can control synthesis options from your HDL source even when the options lack a specific HDL synthesis attribute (such as many of the logic options presented earlier in this chapter). You can also use this attribute to pass entity-level settings and assignments to phases of the Compiler flow beyond Analysis and Synthesis, such as Fitting.

Assignments or settings made through the Quartus II user interface, the Quartus II Settings File (**.qsf**) or the Tcl interface take precedence over assignments or settings made with the `altera_attribute` synthesis attribute in your HDL code.

The syntax for setting this attribute in HDL is the same as the syntax for other synthesis attributes, as shown in "Synthesis Attributes" on page 8–26.

The attribute value is a single string containing a list of Quartus II Settings File variable assignments separated by semicolons, as shown in the following example:

`-name` <*variable_1*> <*value_1*>`;-name` <*variable_2*> <*value_2*>`[;…]`

If the Quartus II option or assignment includes a target, source, and/or section tag, use the following syntax for each Quartus II Settings File variable assignment:

```
-name <variable> <value>
-from <source> -to <target> -section_id <section>
```

The syntax for the full attribute value, including the optional target, source, and section tags for two different Quartus II Settings File assignments is shown in the following example:

```
" -name <variable_1> <value_1> [-from <source_1>] [-to
<target_1>] [-section_id <section_1>]; -name <variable_2>
<value_2> [-from <source_2>] [-to <target_2>] [-section_id
<section_2>] "
```

If a variable's assigned value is a string of text, you must use escaped quotes around the value in Verilog HDL, or double-quotes in VHDL, as in the following examples (using non-existent variable and value terms):

Verilog HDL

```
"VARIABLE_NAME \"STRING_VALUE\""
```

VHDL

```
"VARIABLE_NAME ""STRING_VALUE"""
```

To find the Quartus II Settings File variable name or value corresponding to a specific Quartus II option or assignment, you can make the option setting or assignment in the Quartus II user interface and then note the changes in the QSF. You can also refer to the *Quartus II Settings File Reference Manual*, which documents all variable names.

Example 8–89, 8–90, and 8–91 use `altera_attribute` to set the power-up level of an inferred register. Note that for inferred instances, you cannot apply the attribute to the instance directly, so you should apply the attribute to one of the instance's output nets. The Quartus II software moves the attribute to the inferred instance automatically.

Example 8–89. Verilog-1995 Example of Applying Altera Attribute to an Instance

```
reg my_reg /* synthesis altera_attribute = "-name POWER_UP_LEVEL HIGH" */;
```

Example 8–90. Verilog-2001 Example of Applying Altera Attribute to an Instance

```
(* altera_attribute = "-name POWER_UP_LEVEL HIGH" *) reg my_reg;
```

Example 8–91. VHDL Example of Applying Altera Attribute to an Instance

```
signal my_reg : std_logic;
attribute altera_attribute : string;
attribute altera_attribute of my_reg: signal is "-name POWER_UP_LEVEL
HIGH";
```

Example 8–92, 8–93, and 8–94 use the `altera_attribute` to disable the **Auto Shift Register Replacement** synthesis option for an entity. To apply the Altera Attribute to a VHDL entity, you must set the attribute on its architecture rather than on the entity itself.

Example 8–92. Verilog-1995 Example of Applying Altera Attribute to an Entity

```
module my_entity(…) /* synthesis altera_attribute = "-name
AUTO_SHIFT_REGISTER_RECOGNITION OFF" */;
```

Example 8–93. Verilog-2001 Example of Applying Altera Attribute to an Entity

```
(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION OFF" *)
module my_entity(…) ;
```

Example 8–94. VHDL Example of Applying Altera Attribute to an Entity

```
entity my_entity is
-- Declare generics and ports
end my_entity;
architecture rtl of my_entity is
     attribute altera_attribute : string;
     -- Attribute set on architecture, not entity
     attribute altera_attribute of rtl: architecture is "-name
AUTO_SHIFT_REGISTER_RECOGNITION OFF";
begin
   -- The architecture body
end rtl;
```

You can also use `altera_attribute` for more complex assignments involving more than one instance. In Example 8–95, 8–96, and 8–97, the `altera_attribute` is used to cut all timing paths from `reg1` to `reg2`, equivalent to this Tcl or QSF command:

```
set_instance_assignment -name CUT ON -from reg1 -to reg2
```

Example 8–95. Verilog-1995 Example of Applying Altera Attribute with -to

```
reg reg2;
reg reg1 /* synthesis altera_attribute = "-name CUT ON -to reg2" */;
```

Example 8–96. Verilog-2001 Example of Applying Altera Attribute with -to

```
reg reg2;
(* altera_attribute = "-name CUT ON -to reg2" *) reg reg1;
```

Example 8–97. VHDL Example of Applying Altera Attribute with -to

```
signal reg1, reg2 : std_logic;
attribute altera_attribute: string;
attribute altera_attribute of reg1 : signal is "-name CUT ON -to reg2";
```

You may specify either the `-to` option or the `-from` option in a single `altera_attribute`; integrated synthesis automatically sets the remaining option to the target of the `altera_attribute`. You may also specify wildcards for either option. For example, if you specify "`*`" for the `-to` option instead of `reg2` in these examples, the Quartus II software cuts all timing paths from `reg1` to every other register in this design entity.

The `altera_attribute` can be used only for entity-level settings, and the assignments (including wildcards) apply only to the current entity.

Analyzing Synthesis Results

After you have performed synthesis, you can check your synthesis results in the Analysis and Synthesis Section of the Compilation Report and the Project Navigator.

Analysis and Synthesis Section of the Compilation Report

The Compilation Report, which provides a summary of results for the project, appears after a successful compilation, or you can choose it from the Processing menu. After Analysis and Synthesis, before the Fitter begins, the Summary information provides a summary of utilization based on synthesis data, before Fitter optimizations have occurred. Synthesis-specific information is listed in the **Analysis & Synthesis** section.

There are various report sections under Analysis and Synthesis, including a list of the source files read for the project, the resource utilization by entity after synthesis, and information about state machines, latches, optimization results, and parameter settings.



For more information about each report section, refer to the Quartus II Help.

Project Navigator

The **Hierarchy** tab of the Project Navigator provides a summary of resource information about the entities in the project. After Analysis and Synthesis, before the Fitter begins, the Project Navigator provides a summary of utilization based on synthesis data, before Fitter optimizations have occurred.

If you hold your mouse pointer over one of the entities in the **Hierarchy** tab, a tooltip appears that shows parameter information for each instance.

Analyzing and Controlling Synthesis Messages

This section provides information about the messages generated during synthesis, and how you can control which messages appear during compilation.

Quartus II Messages

The messages that appear during Analysis and Synthesis describe many of the optimizations that the software performs during the synthesis stage, and provide information about how the design is interpreted. You should always check the messages to analyze Critical Warnings and Warnings, because these messages may relate to important design problems. It is also useful to read the information messages Info and Extra Info to get more information about how the software processes your design.

The **Info**, **Extra Info**, **Warning**, **Critical Warning**, and **Error** tabs display messages grouped by type.

You can right-click on a message in the Messages window and get help on the message, locate the source of the message in your design, and manage messages.

You can use message suppression to reduce the number of messages listed after a compilation by preventing individual messages and entire categories of messages from being displayed. For example, if you review a particular message and determine that it is not caused by something in your design that should be changed or fixed, you can suppress the message so it is not displayed during subsequent compilations. This saves time because you see only new messages during subsequent compilations.

You can right-click on an individual message in the Messages window and choose commands in the **Suppress** submenu entry. Alternately, you can open the Message Suppression Manager. To do so, right-click in the Messages window and from the **Suppress** submenu item, click **Message Suppression Manager**.



For more information about messages and suppressing them, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

VHDL and Verilog HDL Messages

The Quartus II software issues a variety of messages when it is analyzing and elaborating the Verilog HDL and VHDL files in your design. These HDL messages are a subset of all Quartus II messages that help you identify potential problems early in the design process.

HDL messages fall into the following three categories:

- **Info message**—Lists a property of your design.
- **Warning message**—Indicates a potential problem in your design. Potential problems come from a variety of sources, including typos, inappropriate design practices, or the functional limitations of your target device. Though HDL warning messages do not always identify actual problems, you should always investigate code that generates an HDL warning. Otherwise, the synthesized behavior of your design might not match your original intent or its simulated behavior.
- **Error message**—Indicates an actual problem with your design. Your HDL code may be invalid due to a syntax or semantic error, or it may not be synthesizable as written. Consult the Help associated with any HDL error messages for assistance in removing the error from your design.

In Example 8–98, the sensitivity list contains multiple copies of the variable `i`. While the Verilog HDL language does not prohibit duplicate entries in a sensitivity list, it is clear that this design has a typo: Variable `j` should be listed on the sensitivity list to avoid a possible simulation/synthesis mismatch.

Example 8–98. Generating an HDL Warning Message

```
//dup.v
module dup(input i, input j, output reg o);
always @ (i or i)
    o = i & j;
endmodule
```

When processing this HDL code, the Quartus II software generates the following warning message:

```
Warning: (10276) Verilog HDL sensitivity list warning
at dup.v(2): sensitivity list contains multiple
entries for "i".
```

In Verilog HDL, variable names are case-sensitive, so the variables my_reg and MY_REG in Example 8–99 are two different variables. However, declaring variables whose names only differ in case may confuse some users, especially those users who use VHDL, where variables are not case-sensitive.

Example 8–99. Generating HDL Info Messages

```
// namecase.v
module namecase (input i, output o);
   reg my_reg;
   reg MY_REG;
   assign o = i;
endmodule
```

When processing this HDL code, the Quartus II software generates the following informational message:

```
Info: (10281) Verilog HDL information at
namecase.v(3): variable name "MY_REG" and variable
name "my_reg" should not differ only in case.
```

In addition, the Quartus II software generates additional HDL info messages to inform you that neither my_reg or MY_REG are used in this small design:

```
Info: (10035) Verilog HDL or VHDL information at
namecase.v(3): object "my_reg" declared but not used
Info: (10035) Verilog HDL or VHDL information at
namecase.v(4): object "MY_REG" declared but not used
```

The Quartus II software allows you to control how many HDL messages you see during the analysis and elaboration of your design files. You can set the HDL Message Level to enable or disable groups of HDL messages, or you can enable or disable specific messages, as described in the following sections.

For more information about synthesis directives and their syntax, refer to "Synthesis Directives" on page 8–29.

Setting the HDL Message Level

The HDL Message Level specifies the types of messages that the Quartus II software displays when it is analyzing and elaborating your design files. Table 8–8 details the information about the HDL message levels.

Table 8–8. HDL Info Message Level

Level	Purpose	Description
Level1	Displays high-severity messages only	If you want to see only those HDL messages that identify likely problems with your design, select **Level1**. When **Level1** is selected, the Quartus II software issues a message only if there is a high probability that it points to an actual problem with your design.
Level2	Displays high-severity and medium-severity messages	If you want to see additional HDL messages that identify possible problems with your design, select **Level2**. This is the default setting.
Level3	Displays all messages, including low-severity messages	If you want to see all HDL info and warning messages, select **Level3**. This level includes extra "LINT" messages that suggest changes to improve the style of your HDL code or make it easier to understand.

You should address all issues reported at the **Level1** setting. The default HDL message level is **Level2**.

To set the HDL Message Level in the user interface, on the Assignments menu, click **Settings**; under **Category**, click **Analysis & Synthesis Settings**. Set the HDL Message Level.

You can override this default setting in a source file with the `message_level synthesis` directive, which takes the values `level1`, `level2`, and `level3`, as shown in Example 8–100 and 8–101.

Example 8–100. Verilog HDL Examples of message_level Directive

```
// altera message_level level1
```

or

```
/* altera message_level level3 */
```

Example 8–101. VHDL Example of message_level Directive

```
-- altera message_level level2
```

A `message_level` synthesis directive remains effective until the end of a file or until the next `message_level` directive. In VHDL, you can use the `message_level` synthesis directive to set the HDL Message Level for entities and architectures, but not for other design units. An HDL Message Level for an entity applies to its architectures, unless overridden by another `message_level` directive. In Verilog HDL, you can use the `message_level` directive to set the HDL Message Level for a module.

Enabling or Disabling Specific HDL Messages by Module/Entity

You can enable or disable a specific HDL info or warning message with its Message ID, which is displayed in parentheses at the beginning of the message. Enabling or disabling a specific message overrides its HDL Message Level. This method is different from the message suppression in the Messages window because you can use this method to disable messages for a specific module or entity. This method applies only the HDL messages, and if you disable a message with this method, the message is listed as a **Suppressed** message in the Quartus II GUI.

To disable specific HDL messages in the GUI, on the Assignments menu, click **Settings**. Select **Analysis & Synthesis Settings** and click the **Advanced** button next to the HDL Message Level setting. In the A**dvanced Message Settings** dialog box, add the Message IDs you wish to enable or disable.

To enable or disable specific HDL messages in your HDL, use the `message_on` and `message_off` synthesis directives. Both directives take a space-separated list of Message IDs. You can enable or disable messages with these synthesis directives immediately before Verilog HDL modules, VHDL entities, or VHDL architectures. You cannot enable or disable a message in the middle of an HDL construct.

A message enabled or disabled via a `message_on` or `message_off` synthesis directive overrides its HDL Message Level or any `message_level` synthesis directive. The message will remain disabled until the end of the source file or until its status is changed by another `message_on` or `message_off` directive.

Example 8–102. Verilog HDL message_off Directive for Message with ID 10000

```
// altera message_off 10000
                    or
/* altera message_off 10000 */
```

Example 8–103. VHDL message_off Directive for Message with ID 10000

```
-- altera message_off 10000
```

Node-Naming Conventions in Quartus II Integrated Synthesis

Being able to find the logic node names after synthesis can be useful during verification or while debugging a design. This section provides an overview of the conventions used by the Quartus II software when it names the nodes created from your HDL design. The section focuses on the conventions for Verilog HDL and VHDL code, but AHDL and BDFs are discussed when appropriate.

Whenever possible, as described in this section, Quartus II integrated synthesis uses wire or signal names from your source code to name nodes such as LEs or ALMs. Some nodes, such as registers, have predictable names that typically do not change when a design is resynthesized, although certain optimizations can affect register names. The names of other nodes, particularly LEs or ALMs that contain only combinational logic, can change due to logic optimizations that the software performs.

This section discusses the following topics:

- "Hierarchical Node-Naming Conventions" on page 8–79
- "Node-Naming Conventions for Registers (DFF or D Flipflop Atoms)" on page 8–80
- "Register Changes During Synthesis" on page 8–81
- "Preserving Register Names" on page 8–84
- "Node-Naming Conventions for Combinational Logic Cells" on page 8–84
- "Preserving Combinational Logic Names" on page 8–86

Hierarchical Node-Naming Conventions

To make each name in the design unique, the Quartus II software adds the hierarchy path to the beginning of each name. The "|" separator is used to indicate a level of hierarchy. For each instance in the hierarchy, the software adds the entity name and the instance name of that entity, using the ":" separator between each entity name and its instance name. For example, if a design instantiates entity `A` with the name `my_A_inst`, the hierarchy path of that entity would be `A:my_A_inst`. The full name of any node is obtained by starting with the hierarchical instance path; followed by a "|", and ending with the node name inside that entity, using the following convention:

<entity 0>:*<instance_name 0>*|*<entity 1>*:
<instance_name 1>|...|*<instance_name n>*

For example, if entity A contains a register (DFF atom) called `my_dff`, its full hierarchy name would be `A:my_A_inst|my_dff`.

On the **Compilation Process Settings** page of the **Settings** dialog box, click **More Settings** and turn off **Display entity name for node name** to instruct the Compiler to generate node names that do not contain the name for each level of the hierarchy. With this option off, the node names use the following convention:

<instance_name 0>|<instance_name 1>|...|<instance_name n>

Node-Naming Conventions for Registers (DFF or D Flipflop Atoms)

In Verilog HDL and VHDL, inferred registers are named after the `reg` or `signal` connected to the output.

Example 8–104 is a description of a register in Verilog HDL that creates a DFF primitive called `my_dff_out`:

Example 8–104. Verilog HDL Register

```
wire dff_in, my_dff_out, clk;

always @ (posedge clk)
  my_dff_out <= dff_in;
```

Similarly, Example 8–105 is a description of a register in VHDL that creates a DFF primitive called `my_dff_out`.

Example 8–105. VHDL Register

```
signal dff_in, my_dff_out, clk;
process (clk)
  begin
  if (rising_edge(clk)) then
    my_dff_out <= dff_in;
  end if;
end process;
```

In AHDL designs, DFF registers are declared explicitly rather than inferred, so the software uses the user-declared name for the register.

For schematic designs using BDF, all elements are given a name when they are instantiated in the design, so the software uses the user-defined name for the register or DFF.

In the special case that a wire or signal (such as `my_dff_out` in the preceding examples) is also an output pin of your top-level design, the Quartus II software cannot use that name for the register (for example,

cannot use my_dff_out) because the software requires that all logic and I/O cells have unique names. In this case, the Quartus II integrated synthesis appends ~reg0 to the register name.

For example, the Verilog HDL code in Example 8–106 produces a register called q~reg0:

Example 8–106. Verilog HDL Register Feeding Output Pin

```
module my_dff (input clk, input d, output q);
  always @ (posedge clk)
    q <= d;
endmodule
```

This situation occurs only for registers driving top-level pins. If a register drives a port of a lower level of the hierarchy, the port is removed during hierarchy flattening and the register retains its original name, in this case, q.

Register Changes During Synthesis

On some occasions, you may not be able to find registers that you expect to see in the synthesis netlist. Registers may be removed by logic optimization, or their names may be changed due to synthesis optimization. Common optimizations include inference of a state machine, counter, adder-subtractor, or shift register from registers and surrounding logic. Other common register changes occur when registers are packed into dedicated hardware on the FPGA, such as a DSP block or a RAM block.

This section describes the following factors that can affect register names:

- "Synthesis and Fitting Optimizations" on page 8–82
- "State Machines" on page 8–83
- "Inferred Adder-Subtractors, Shift Registers, Memory, and DSP Functions" on page 8–83
- "Packed Input and Output Registers of RAM and DSP Blocks" on page 8–83
- "Preserving Register Names" on page 8–84
- "Preserving Combinational Logic Names" on page 8–86

Synthesis and Fitting Optimizations

Registers may be removed by synthesis logic optimization if they are not connected to inputs or outputs in the design, or if the logic can be simplified due to constant signal values. Register names may also be changed due to synthesis optimizations, such as when duplicate registers are merged together to reduce resource utilization.

NOT-gate push back optimizations may affect registers that use preset signals. This type of optimization can impact your timing assignments when registers are used as clock dividers. If this situation occurs in your design, change the clock settings to work on the new register name.

Synthesis netlist optimizations often change node names because registers may be combined or duplicated to optimize the design.



For more information about the type of optimizations performed by synthesis netlist optimizations, refer to the *Netlist Optimization and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

The Quartus II **Compilation Report** provides a list of registers that are removed during synthesis optimizations, and a brief reason for the removal. In the **Analysis & Synthesis** folder, open **Optimization Results**, and then open **Register Statistics**, and click on the **Registers Removed During Synthesis** report, and the **Removed Registers Triggering Further Register Optimizations** report. The second report contains a list of registers that are the cause of other registers being removed in the design. It provides a brief reason for the removal, and a list of registers that were removed due to the removal of the initial register.

Synthesis creates synonyms for registers duplicated with the **Maximum Fan-Out** option (or `maxfan` attribute). Therefore, timing assignment applied to nodes that are duplicated with this option are applied to the new nodes as well.

The Quartus II Fitter can also change node names after synthesis (for example, when the Fitter uses register packing to pack a register into an I/O element, or when logic is modified by physical synthesis). The Fitter creates synonyms for duplicated registers so that timing analysis can use the existing node name when applying assignments.

You can instruct the Quartus II software to preserve certain nodes throughout compilation so that you can use them for verification or making assignments. For more information, refer to "Preserving Register Names" on page 8–84.

State Machines

If a state machine is inferred from your HDL code, the registers that represent the states are mapped into a new set of registers that implement the state machine. Most commonly, the software converts the state machine into a one-hot form where each state is represented by one register. In this case, for Verilog HDL or VHDL designs, the registers are named according to the name of the state register and the states, where possible.

For example, consider a Verilog HDL state machine where the states are `parameter state0 = 1, state1 = 2, state2 = 3`, and where the state machine register is declared as `reg [1:0] my_fsm`. In this example, the three one-hot state registers are named `my_fsm.state0`, `my_fsm.state1`, and `my_fsm.state2`.

In AHDL, state machines are explicitly specified with a machine name. State machine registers are given synthesized names based on the state machine name but not the state names. For example, if a state machine is called `my_fsm` and has four state bits, they may be synthesized with names such as `my_fsm~12`, `my_fsm~13`, `my_fsm~14`, and `my_fsm~15`.

Inferred Adder-Subtractors, Shift Registers, Memory, and DSP Functions

The Quartus II software infers megafunctions from Verilog HDL and VHDL code for logic that forms adder-subtractors, shift registers, RAM, ROM, and arithmetic functions that can be placed in DSP blocks.



For information about inferring megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Because adder-subtractors are part of a megafunction instead of generic logic, the combinational logic exists in the design with different names. For shift registers, memory, and DSP functions, the registers and logic are typically implemented inside the dedicated RAM or DSP blocks in the device. Thus, the registers are not visible as separate LEs or ALMs.

Packed Input and Output Registers of RAM and DSP Blocks

Registers can be packed into the input registers and output registers of RAM and DSP blocks, so that they are not visible as separate registers in LEs or ALMs.



For information about packing registers into RAM and DSP megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Preserving Register Names

You may want to preserve certain register names for verification or debugging, or to ensure that timing assignments are applied correctly. Quartus II integrated synthesis preserves certain nodes automatically if they are likely to be used in a timing constraint.

Use the `preserve` attribute to instruct the Compiler not to minimize or remove a specified register during synthesis optimizations or register netlist optimizations. Refer to "Preserve Registers" on page 8–44 for details.

Use the `noprune` attribute to preserve a fan-out-free register through the entire compilation flow. Refer to "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 8–46 for details.

Use synthesis attribute `syn_dont_merge` to make sure registers are not merged with other registers, and other registers are not merged with it. Refer to "Disable Register Merging/Don't Merge Register" on page 8–45 for details.

Node-Naming Conventions for Combinational Logic Cells

Whenever possible for Verilog HDL, VHDL, and AHDL code, the Quartus II software uses wire names that are the targets of assignments, but may change the node names due to synthesis optimizations.

For example, consider the Verilog HDL code in Example 8–107. Quartus II integrated synthesis uses the names `c`, `d`, `e`, and `f` for the combinational logic cells that are produced.

Example 8–107. Naming Nodes for Combinational Logic Cells in Verilog HDL

```
wire c;
reg d, e, f;

assign c = a | b;
always @ (a or b)
    d = a & b;
always @ (a or b) begin : my_label
    e = a ^ b;
end

always @ (a or b)
    f = ~(a | b);
```

For schematic designs using BDF, all elements are given a name when they are instantiated in the design and the software uses the user-defined name when possible.



Node naming conventions for schematic buses in the Quartus II software version 7.2 and later are different than the MAX+PLUS II software and older versions of the Quartus II software. In most cases, the Quartus II software uses the appropriate naming convention for the design source file. For designs created using the Quartus II software version 7.1 or earlier, it uses the MAX+PLUS II naming convention. For designs created in the Quartus II software version 7.2 and later, it uses the Quartus II naming convention that matches the behavior of standard HDLs. In some cases, however, a design may contain files created in various versions. To set an assignment for a particular instance in the Assignment Editor, enter the instance name in the **To** field, choose **Block Design Naming** from the Assignment Name list, and set the value to **MaxPlusII** or **QuartusII**.

If logic cells, such as those created in Example 8–107, are packed with registers in device architectures such as the Stratix and Cyclone device families, those names may not appear in the netlist after fitting. In other devices, such as newer families in the Stratix and Cyclone series device families, the register and combinational nodes are kept separate throughout the compilation, so these names are more often maintained through fitting.

When logic optimizations occur during synthesis, it is not always possible to retain the initial names as described. In some cases, synthesized names will be used, which are the wire names with a tilde (~) and a number appended. For example, if a complex expression is assigned to a wire `w` and that expression generates several logic cells, those cells may have names such as `w`, `w~1`, `w~2`, and so on. Sometimes the original wire name `w` is removed, and an arbitrary name such as `rtl~123` is created. It is a goal of Quartus II integrated synthesis to retain user names whenever possible. Any node name ending with ~<*number*> is a name created during synthesis, which may change if the design is changed and re-synthesized. Knowing these naming conventions can help you understand your post-synthesis results and make it easier to debug your design or make assignments.

The software maintains combinational clock logic by making sure nodes that are likely to be a clock don't get changed during synthesis. The software also maintains (or "protects") multiplexers in clock trees so that the TimeQuest Timing Analyzer has information about which paths are unate, to allow complete and correct analysis of combinational clocks.

Multiplexers often occur in clock trees when the design selects between different clocks. To help analysis of clock trees, the software ensures that each multiplexer encountered in a clock tree is broken into 2:1 multiplexers, and each of those 2:1 multiplexers is mapped into one look-up table (independent of the device family). This optimization might result in a slight increase in area, and for some designs a decrease in timing performance. You can turn off this multiplexer protection with the option **Clock MUX Protection** under **More Settings** on the **Analysis & Synthesis** page of the **Settings** dialog box. This option applies to Arria GX devices, the Stratix and Cyclone series, and MAX II devices.

Preserving Combinational Logic Names

You may want to preserve certain combinational logic node names for verification or debugging, or to ensure that timing assignments are applied correctly.

Use the `keep` attribute to keep a wire name or combinational node name through logic synthesis minimizations and netlist optimizations. Refer to "Keep Combinational Node/Implement as Output of Logic Cell" on page 8–47 for details.

For any internal node in your design clock network, use `keep` to protect the name so that you can apply correct clock settings. Also, set the attribute on combinational logic involved in `cut` assignments and `-through` assignments.



Setting the `keep` attribute on combinational logic may increase the area utilization and increase the delay of the final mapped logic because it requires the insertion of extra combinational logic. Use the attribute only when necessary.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF form.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

You can specify many of the options described in this section either on an instance, global level, or both.

Use the following Tcl command to make a global assignment:

```
set_global_assignment -name <QSF Variable Name> <Value>
```

Use the following Tcl command to make an instance assignment:

```
set_instance_assignment -name <QSF Variable Name> <Value>\
-to <Instance Name>
```

Adding an HDL File to a Project and Setting the HDL Version

Use the following Tcl assignments to add an HDL or schematic entry design file to your project:

```
set_global_assignment -name VERILOG_FILE <file name>.<v|sv>
set_global_assignment -name SYSTEMVERILOG_FILE <file name>.sv
set_global_assignment -name VHDL_FILE <file name>.<vhd|vhdl>
set_global_assignment -name AHDL_FILE <file name>.tdf
set_global_assignment -name BDF_FILE <file name>.bdf
```



You can use any file extension for design files, as long as you specify the correct language when adding the design file. For example, you can use **.h** for Verilog header files.

To specify the Verilog HDL or VHDL version, use the following option at the end of the `VERILOG_FILE` or `VHDL_FILE` command:

```
-HDL_VERSION <language version>
```

The variable <*language version*> takes one of the following values:

- VERILOG_1995
- VERILOG_2001
- SYSTEMVERILOG_2005
- VHDL87
- VHDL93

For example, to add a Verilog HDL file called **my_file** that is written in Verilog-1995, use the following command:

```
set_global_assignment -name VERILOG_FILE my_file.v -HDL_VERSION VERILOG_1995
```

Quartus II Synthesis Options

Table 8–9 lists the Quartus II Settings File variable names and applicable values for the settings discussed in this chapter. The Quartus II Settings File variable name is used in the Tcl assignment to make the setting along with the appropriate value. The *Type* column indicates whether the setting is supported as a Global setting, or an Instance setting, or both.

Table 8–9. Quartus II Synthesis Options (Part 1 of 2)

Setting Name	Quartus II Settings File Variable	Values	Type
Allow Any RAM Size for Recognition	ALLOW_ANY_RAM_SIZE_FOR_RECOGNITION	ON, OFF	Global, Instance
Allow Any ROM Size for Recognition	ALLOW_ANY_ROM_SIZE_FOR_RECOGNITION	ON, OFF	Global, Instance
Allow Any Shift Register Size for Recognition	ALLOW_ANY_SHIFT_REGISTER_SIZE_FOR_RECOGNITION	ON, OFF	Global, Instance
Auto DSP Block Replacement	AUTO_DSP_RECOGNITION	ON, OFF	Global, Instance
Auto RAM Replacement	AUTO_RAM_RECOGNITION	ON, OFF	Global, Instance
Auto ROM Replacement	AUTO_ROM_RECOGNITION	ON, OFF	Global, Instance
Auto Shift-Register Replacement	AUTO_SHIFT_REGISTER_RECOGNITION	ON, OFF	Global, Instance
Block Design Naming	BLOCK_DESIGN_NAMING	AUTO, MAXPLUSII, QUARTUSII	Global, Instance
Fast Input Register	FAST_INPUT_REGISTER	ON, OFF	Instance
Fast Output Enable Register	FAST_OUTPUT_ENABLE_REGISTER	ON, OFF	Instance
Fast Output Register	FAST_OUTPUT_REGISTER	ON, OFF	Instance
Implement as Output of Logic Cell	IMPLEMENT_AS_OUTPUT_OF_LOGIC_CELL	ON, OFF	Instance
Disable Register Merging	DONT_MERGE_REGISTER	ON, OFF	Instance

Table 8–9. Quartus II Synthesis Options (Part 2 of 2)

Setting Name	Quartus II Settings File Variable	Values	Type
Maximum Fan-Out	MAX_FANOUT	<*Maximum Fan-Out Value*>	Instance
Optimization Technique	<*device family*>_OPTIMIZATION_TECHNIQUE	Area, Speed, Balanced	Global, Instance
PowerPlay Power Optimization	OPTIMIZE_POWER_DURING_SYNTHESIS	"NORMAL COMPILATION", "EXTRA EFFORT", OFF	Global, Instance
Power-Up Don't Care	ALLOW_POWER_UP_DONT_CARE	ON, OFF	Global
Power-Up Level	POWER_UP_LEVEL	HIGH, LOW	Instance
Preserve Registers	PRESERVE_REGISTER	ON, OFF	Instance
Remove Duplicate Logic	REMOVE_DUPLICATE_LOGIC	ON, OFF	Global, Instance
Remove Duplicate Registers	REMOVE_DUPLICATE_REGISTERS	ON, OFF	Global, Instance
Remove Redundant Logic Cells	REMOVE_REDUNDANT_LOGIC_CELLS	ON, OFF	Global
Restructure Multiplexers	MUX_RESTRUCTURE	On, Off, Auto	Global, Instance
Speed Optimization Technique for Clock Domains	SYNTH_CRITICAL_CLOCK	ON, OFF	Instance
State Machine Processing	STATE_MACHINE_PROCESSING	AUTO, "MINIMAL BITS", "ONE HOT", "USER-ENCODED"	Global, Instance

Assigning a Pin

Use the following Tcl command to assign a signal to a pin or device location.

```
set_location_assignment -to <signal name> <location>
```

For example,

```
set_location_assignment -to data_input Pin_A3
```

Valid locations are pin location names. Some device families also support edge and I/O bank locations. Edge locations are EDGE_BOTTOM, EDGE_LEFT, EDGE_TOP, and EDGE_RIGHT. I/O bank locations include IOBANK_1 to IOBANK_n, where n is the number of I/O banks in a particular device.

Creating Design Partitions for Incremental Compilation

To create a partition, use the following command:

```
set_instance_assignment -name PARTITION_HIERARCHY \
<file name> -to <destination> -section_id <partition name>
```

The <*destination*> should be the entity's short hierarchy path. A short hierarchy path is the full hierarchy path without the top-level name, for example: "ram:ram_unit|altsyncram:altsyncram_component" (with quotation marks). For the top-level partition, you can use the pipe (|) symbol to represent the top-level entity.

For more information about hierarchical naming conventions, refer to "Node-Naming Conventions in Quartus II Integrated Synthesis" on page 8–79.

The <*partition name*> is the user-designated partition name, which must be unique and less than 1024 characters long. The name can consist only of alpha-numeric characters, as well as pipe (|), colon (:), and underscore (_) characters. Altera recommends enclosing the name in double quotation marks (" ").

The <*file name*> is the name used for internally generated netlists files during incremental compilation. Netlists are named automatically by the Quartus II software based on the instance name if you create the partition in the user interface. If you are using Tcl to create your partitions, you must assign a custom file name that is unique across all partitions. For the top-level partition, the specified file name is ignored, and you can use any dummy value. To ensure the names are safe and platform independent, file names must be unique regardless of case. For example, if a partition uses the file name my_file, no other partition can use the file name MY_FILE. For simplicity, Altera recommends that you base each file name on the corresponding instance name for the partition.

The software stores all netlists in the **db** compilation database directory.

Conclusion

The Quartus II software includes complete Verilog HDL and VHDL language support, as well as support for Altera-specific languages, making it an easy-to-use, standalone solution for Altera designs. You can use the synthesis options available in the software to help you improve your synthesis results, giving you more control over the way your design is synthesized. Use Quartus II reports and messages to analyze your compilation results.

Referenced Documents

This chapter references the following documents:

- Assignment Editor chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Designing With Low-Level Primitives User Guide*
- *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*
- *Introduction to the Quartus II Software*
- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Netlist Optimization and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Reference Manual*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 8–10 shows the revision history for this chapter.

Table 8–10. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Added three new constructs to "SystemVerilog Support" on page 8–7 • Added new section "State Machine Editor" on page 8–13 • Renamed section as "Analyzing and Controlling Synthesis Messages" on page 8–74 and added section "Quartus II Messages" on page 8–74 • Other minor changes and text additions	Updated for Quartus II software version 7.2.
May 2007 v7.1.0	• Updated language constraints supported in "SystemVerilog Support" on page 8–7 • Updated "Incremental Synthesis and Incremental Compilation" on page 8–23 • Removed Preserve Hierarchical Boundary section and replaced it with updated section "Partitions for Preserving Hierarchical Boundaries" on page 8–23 • Updated "Synthesis Attributes" on page 8–26 • Added "Disable Register Merging/Don't Merge Register" on page 8–45 • Added "Don't Retime, Disabling Synthesis Netlist Optimizations" on page 8–48 • Added "Don't Replicate, Disabling Synthesis Netlist Optimizations" on page 8–49 • Updated and added more description to "Node-Naming Conventions in Quartus II Integrated Synthesis" on page 8–79 • Added "Preserving Register Names" on page 8–84 • Added "Preserving Combinational Logic Names" on page 8–86 • Updated "Adding an HDL File to a Project and Setting the HDL Version" on page 8–88 • Updated Table 8–9 on page 8–89 to match the new chapter content • Added "Referenced Documents" on page 8–92 • Added Arria GX devices where appropriate	Updates made for new attributes, options, and language support in the Quartus II software version 7.1 and Arria GX devices.
March 2007 v7.0.0	Updated date and revision for the Quartus II software version 7.0.	—

November 2006 v6.1.0	• Added information on how to set the HDL version in "Verilog HDL Support" on page 8–5 and "VHDL Support" on page 8–10 • Updated the list of supported constructs in "SystemVerilog Support" on page 8–7 • Added "Initial Constructs and Memory System Tasks" on page 8–8 • Added "Design Libraries" on page 8–13 to include information on libraries and duplicate entity names in all languages • Added "Using Parameters/Generics" on page 8–18 • Reorganized the options in the Quartus II Synthesis Options section • Added information about reset status to "State Machine Processing" on page 8–33 • Added "Safe State Machines" on page 8–36 • Removed section on obsoleted logic option Remove Duplicate Logic • Added "Controlling Clock Enable Signals with Auto Clock Enable Replacement & syn_direct_enable" on page 8–46 • Added "RAM to Logic Cell Conversion" on page 8–49 • Added "Turning off Add Pass-Through Logic to Inferred RAMs/ no_rw_check" on page 8–51 • Added synthesis_off and on directives to "Translate Off and On / Synthesis Off and On" on page 8–59 and "Ignore translate_off and synthesis_off Directives" on page 8–60 • Updated options to include Stratix III in the Stratix series of devices as required	This chapter has been updated to include information about additional functionality and support for integrated synthesis. The updates made to this chapter describe new and/or enhanced features to language support, incremental synthesis, and many of the Quartus II synthesis options.
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0: • Added language support. • Added Quartus II Synthesis options. • Added information on setting other Quartus II options in HDL source code.	—
December 2005 v5.1.1	Minor typographic update.	—
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 7 was formerly Chapter 8 in version 5.0.	—
May 2005 v5.0.0	• Chapter 8 was formerly Chapter 6 in version 4.2. • Updated information. • Updated figures. • Restructured information. • Renamed sections. • New functionality for the Quartus II software 5.0.	—
December 2004 v3.0	• Chapter 7 was formerly Chapter 8 in version 4.1. • Added documentation of incremental synthesis feature • New functionality for the Quartus II software version 4.2	—

June 2004 v2.0	• Updates to tables, figures. • New functionality for the Quartus II software version 4.1.	—
Feb. 2004 v1.0	Initial release.	—

9. Mentor Graphics LeonardoSpectrum Support

QII51010-7.2.0

Introduction

As programmable logic devices (PLDs) become more complex and require increased performance, advanced synthesis has become an important part of the design flow. Combining HDL coding techniques, Mentor Graphics LeonardoSpectrum™ software constraints, and Quartus® II options provide the performance increase needed for today's system-on-a-programmable-chip (SOPC) designs.

The LeonardoSpectrum software is a mature synthesis tool supporting legacy devices and many current devices. The LeonardoSpectrum software version 2007a supports the Stratix® III, Stratix II, Stratix, Stratix GX, Cyclone™ II, Cyclone, MAX® II, MAX series, APEX™ series, FLEX® series, and ACEX® series device families. Altera® recommends using the advanced Precision Synthesis software for new designs in new device families.

For more information about Precision RTL Synthesis, refer to the *Mentor Graphics Precision RTL Synthesis Support* chapter in volume 1 of the *Quartus II Handbook*.

This chapter documents key design methodologies and techniques for achieving better performance in Altera devices using the LeonardoSpectrum and Quartus II design flow.



This chapter assumes that you have set up, licensed, and are familiar with the LeonardoSpectrum software.

To obtain and license the LeonardoSpectrum software, refer to the Mentor Graphics website at **www.mentor.com**. For information about installing the LeonardoSpectrum software and setting up your working environment, refer to the *LeonardoSpectrum Installation Guide* and the *LeonardoSpectrum User's Manual*.

Design Flow

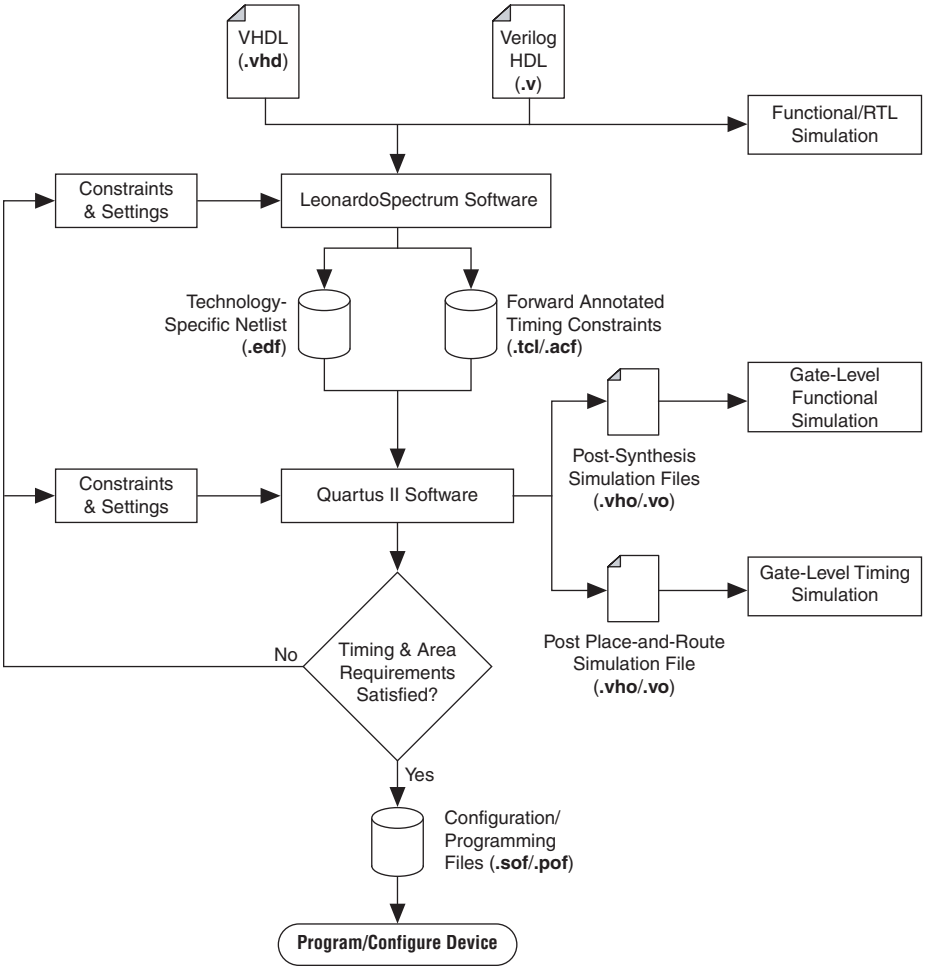
Following are the basic steps in a LeonardoSpectrum-Quartus II design flow:

1. Create Verilog HDL or VHDL design files in the LeonardoSpectrum software or a text editor.
2. Import the Verilog HDL or VHDL design files into the LeonardoSpectrum software for synthesis.
3. Select a target device and add timing constraints and compiler directives to help optimize the design during synthesis.
4. Synthesize the project in the LeonardoSpectrum software.
5. Create a Quartus II project and import the technology-specific EDIF Input File (.**edf**) netlist and the Tcl Script File (.**tcl**) generated by the LeonardoSpectrum software into the Quartus II software for placement and routing, and for performance evaluation.
6. After obtaining place-and-route results that meet your needs, configure or program the Altera device.

Figure 9–1 shows the recommended design flow using the LeonardoSpectrum and Quartus II software.

If your area and timing requirements are satisfied, use the programming files generated from the Quartus II software to program or configure the Altera device. As shown in Figure 9–1, if the area or timing requirements are not met, change the constraints in the LeonardoSpectrum software and re-run the synthesis. Repeat the process until the area and timing requirements are met. You can also use other Quartus II software options and techniques to meet the area and timing requirements.

Figure 9–1. Recommended Design Flow Using LeonardoSpectrum and Quartus II Software



The LeonardoSpectrum software supports both VHDL and Verilog HDL source files. With the appropriate license, it also supports mixed synthesis, allowing a combination of VHDL and Verilog HDL source

files. After synthesis, the LeonardoSpectrum software produces several intermediate and output files. Table 9–1 lists these file extensions with a short description of each file.

Table 9–1. LeonardoSpectrum Intermediate and Output Files

File Extension(s)	File Description
.xdb	Technology-independent register transfer level (RTL) netlist file that can only be read by the LeonardoSpectrum software.
.edf	Technology-specific output netlist in electronic design interchange format (EDIF).
.acf/**.tcl** *(1)*	Forward-annotated constraint file containing constraints and assignments.

Note to Table 9–1:

(1) An assignment and configuration (**.acf**) file is created only for ACEX 1K, FLEX series, and MAX series devices. The assignment and configuration file is generated for backward compatibility with the MAX+PLUS® II software. A Tcl Script File (**.tcl**) is generated for the Quartus II software which also contains Tcl commands to create a Quartus II project.



Altera recommends that you do not use project directory names that include spaces. Some file operations in the LeonardoSpectrum software do not work correctly if the path name contains spaces.

Specify timing constraints and compiler directives for the design in the LeonardoSpectrum software, or in a constraint file (**.ctr**). Many of these constraints are forward-annotated in the Tcl file for use by the Quartus II software.

The LeonardoInsight™ Schematic Viewer is an add-on graphical tool for schematic views of the technology-independent RTL netlist (**.xdb**) and the technology-specific gate-level results. You can use the Schematic Viewer to visually analyze and debug the design. It also supports cross probing between the RTL and gate-level schematics, the design browser, and the source code in the HDLInventor™ text editor.

Optimization Strategies

You can configure most general settings in the **Quick Setup** tab in the LeonardoSpectrum user interface. Other Flow tabs provide additional options, and some Flow tabs include multiple Power tabs (at the bottom of the screen) with still more options. Advanced optimization options in the LeonardoSpectrum software include timing-driven synthesis, encoding style, resource sharing, and mapping I/O registers.

Timing-Driven Synthesis

The LeonardoSpectrum software supports timing-driven synthesis through user-assigned timing constraints to optimize the performance of the design. Setting constraints in the LeonardoSpectrum software are straightforward. Constraints such as clock frequency can be specified globally or for individual clock signals. The following sections describe how to set the various types of timing constraints in the LeonardoSpectrum software.

The timing constraints described in the "Global Power Tab" section are set in the **Constraints** Flow tab. In this tab, there are Power tabs at the bottom, such as **Global** and **Clock**, for setting various constraints.

Global Power Tab

The **Global** tab is the default Power tab in the **Constraints** Flow tab. Specify the global clock frequency here. The **Clock Frequency** on the **Quick Setup** tab is equivalent to the **Registers to Registers** delay setting. You can also specify the following: **Input Ports to Registers**, **Registers to Output Ports**, and **Inputs to Outputs** delays that correspond to global t_{SU}, t_{CO}, and t_{PD} requirements, respectively, in the Quartus II software. The timing diagram on this tab reflects the settings you have made.

Clock Power Tab

You can set various constraints for each clock in your design. First, select the clock name in the **Clock(s)** window. The clock names appear after the design is read from the **Input** Flow tab. Configure settings for that particular clock and click **Apply**. If necessary, you can also set the **Duty Cycle** to a value other than the default 50%. The timing diagram shows these settings.

If a clock has an **Offset** from the main clock, which is considered to be time "`0`", this constraint corresponds to the `OFFSET_FROM_BASE_CLOCK` setting in the Quartus II software.

You can specify the pin number for the clock input pin in the **Pin Location** field. This pin number is passed to the Quartus II software for place-and-route, but does not affect synthesis in the LeonardoSpectrum software.

Input and Output Power Tabs

Configure settings for individual input or output pins in the **Input** and **Output** tabs. First, select a name in the **Input Ports** or **Output Ports** window. The names appear after the design is read from the **Input** Flow tab. Then make the setting for that pin as described below.

The **Arrival Time** setting indicates that the input signal arrives a specified time after the rising clock edge (time "0"). This setting constrains the path from the pin to the first register by including the arrival time in the total delay, and corresponds to the `EXTERNAL_INPUT_DELAY` assignment in the Quartus II software.

The **Required Time** setting indicates the maximum delay after time "0" that the output signal should arrive at the output pin. This setting directly constrains the register to output delay, and corresponds with the `EXTERNAL_OUTPUT_DELAY` assignment in the Quartus II software.

Specify the pin number for the I/O pin in the **Pin Location** field. This pin number is passed to the Quartus II software for place-and-route, but does not affect synthesis in the LeonardoSpectrum software.

Other Constraints

The following sections describe other constraints that can be set with the LeonardoSpectrum user interface.

Encoding Style

The LeonardoSpectrum software encodes state machines during the synthesis process. To improve performance when coding state machines, separate state machine logic from all arithmetic functions and data paths. Once encoded, a design cannot be re-encoded later in the optimization process. You must follow a particular VHDL or Verilog HDL coding style for the LeonardoSpectrum software to identify the state machine.

Table 9–2 shows the state machine encoding styles supported by the LeonardoSpectrum software.

Table 9–2. State Machine Encoding Styles in the LeonardoSpectrum Software

Style	Description
Binary	Generates state machines with the fewest possible flipflops. Binary state machines are useful for area-critical designs when timing is not the primary concern.
Gray	Generates state machines where only one flipflop changes during each transition. Gray-encoded state machines tend to be glitchless.
One-hot	Generates state machines containing one flipflop for each state. One-hot state machines provide the best performance and shortest clock-to-output delays. However, one-hot implementations are usually larger than binary implementations.
Random	Generates state machines using random state machine encoding. Only use random state machine encoding when no other implementation achieves the desired results.
Auto (default)	Implements binary or one-hot encoding, depending on the size of enumerated types in the state machine.

The **Encoding Style** setting is created in the **Input** Flow tab. It instructs the software to use a particular state machine encoding style for all state machines. The default **Auto** selection implements binary or one-hot encoding, depending on the size of enumerated types in the state machine.



To ensure proper recognition and improve performance when coding state machines, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook* for design guidelines.

Resource Sharing

You can also enable the **Resource Sharing** setting in the **Input** Flow tab. This setting allows optimization to reduce device resources. You should generally leave this setting turned on.

Mapping I/O Registers

The **Map I/O Registers** option is located in the **Technology** Flow tab. The **Map I/O Registers** option applies to Altera FPGAs containing I/O cells (IOCs) or I/O elements (IOE). If the option is turned on, input or output registers are moved into the device's I/O cells for faster setup or clock-to-output times.

Timing Analysis with the Leonardo-Spectrum Software

The LeonardoSpectrum software reports successful synthesis with an information message in the **Transcript** or **Information** window. Estimated device usage and timing results are reported in the Device Utilization section of this window. Figure 9–2 shows an example of a LeonardoSpectrum compilation report.

Figure 9–2. LeonardoSpectrum Compilation Report

```
************************************************
Device Utilization for EP20K200EQC208
************************************************
Resource                 Used    Avail    Utilization
------------------------------------------------------
IOs                      22      136       16.18%
LCs                      114     8320       1.37%
Memory Bits              0       106496     0.00%


------------------------------------------------
                         Clock Frequency Report

        Clock                 : Frequency
      ------------------------------------


        clk                   : 52.2 MHz
        clk2                  : 149.5 MHz

                         Critical Path Report
```

The LeonardoSpectrum software estimates the timing results based on timing models. The LeonardoSpectrum software has no information about how the design is placed and routed in the Quartus II software, so it cannot report accurate routing delays. Additionally, if the design includes any black-boxed Altera-specific functions, the LeonardoSpectrum software does not report timing information for these functions.

Final timing results are generated by the Quartus II software and are reported separately in the **Transcript** or **Information** window if the **Run Integrated Place and Route** option is turned on. Refer to "Integration with the Quartus II Software" on page 9–10 for more information.

Exporting Designs Using NativeLink Integration

You can use NativeLink® integration to integrate the LeonardoSpectrum software and the Quartus II software with a single GUI for both the synthesis and place-and-route operations. NativeLink integration allows you to run the Quartus II software from within the LeonardoSpectrum software GUI, or to run the LeonardoSpectrum software from within the Quartus II software GUI for device families supported in the Quartus II software.

Generating Netlist Files

The LeonardoSpectrum software generates an EDIF netlist file readable as an input file in the Quartus II software for place-and-route. Select the EDIF file option name in the **Output** Flow tab. The EDIF netlist is also generated if the **Auto** option is turned on in the **Output** Flow tab.

Including Design Files for Black-Boxed Modules

If the design has black-boxed megafunctions, be sure to include the MegaWizard® Plug-In Manager-generated custom megafunction variation design file in the Quartus II project directory, or add it to the list of project files for place-and-route.

Passing Constraints with Scripts

The LeonardoSpectrum software can write out a Tcl file called <***project name***>**.tcl**. This file contains commands to create a Quartus II project along with constraints and other assignments. To output a Tcl script, turn on the **Write Vendor Constraint Files** option in the **Output** Flow tab.

To create and compile a Quartus II project using the Tcl file generated from the LeonardoSpectrum software, perform the following steps in the Quartus II software:

1. Place the EDIF netlist files and Tcl scripts in the same directory.
2. On the View menu, point to Utility, and click **Tcl Console** to open the Quartus II Tcl Console.
3. Type `source <path>/<project name>.tcl` ↵, at a **Tcl Console** command prompt.
4. On the File menu, click **Open Project** to open the new project. On the Processing menu, click **Start Compilation**.

Integration with the Quartus II Software

The **Place And Route** section in the **Quick Setup** tab allows you to launch the Quartus II software from within the LeonardoSpectrum software. Turn on the **Run Integrated Place and Route** option to start the compilation using the Quartus II software to show the fitting and performance results. You can also run the place-and-route software by turning on the **Run Quartus** option on the **Physical** Flow tab and clicking **Run PR**.

To use integrated place-and-route software, on the Options menu, point to Place and Route Path and click **Tools**, and specify the location of the Quartus II software executable file (browse to <*Quartus II software installation directory*>**/bin)**.

Guidelines for Altera Megafunctions and LPM Functions

Altera provides parameterizable megafunctions ranging from simple arithmetic units, such as adders and counters, to advanced phase-locked loop (PLL) blocks, multipliers, and memory structures. These functions are performance-optimized for Altera devices. Megafunctions include the library of parameterized modules (LPM), device-specific megafunctions such as PLLs, LVDS, and digital signal processing (DSP) blocks, intellectual property (IP) available as Altera MegaCore® functions, and IP available through the Altera Megafunction Partners Program (AMPP^{sm}).



Some IP cores require that you synthesize them in the LeonardoSpectrum software. Refer to the user guide for the specific IP.

There are two methods for handling megafunctions in the LeonardoSpectrum software: inference and instantiation.

The LeonardoSpectrum software supports inferring some of the Altera megafunctions, such as multipliers, DSP functions, and RAM and ROM blocks. The LeonardoSpectrum software supports all Altera megafunctions through instantiation.

Instantiating Altera Megafunctions

There are two methods of instantiating Altera megafunctions in the LeonardoSpectrum software. The first and least common method is to directly instantiate the megafunction in the Verilog HDL or VHDL code. The second method, to maintain target technology awareness, is to use the MegaWizard Plug-In Manager in the Quartus II software to setup and parameterize a megafunction variation. The megafunction wizard creates a wrapper file that instantiates the megafunction. The advantage of using the megafunction wizard in place of the instantiation method is the

megafunction wizard properly sets all the parameters and you do not need the library support required in the direct instantiation method. This is referred to as the black box methodology.



Altera recommends using the megafunction wizard to ensure that the ports and parameters are set correctly.



When directly instantiating megafunctions, see the Quartus II Help for a list of the ports and parameters.

Inferring Altera Memory Elements

The LeonardoSpectrum software can infer memory blocks from Verilog HDL or VHDL code. When the LeonardoSpectrum software detects a RAM or ROM from the style of the RTL code at a technology-independent level, it then maps the element to a generic module in the RTL database. During the technology-mapping phase of synthesis, the LeonardoSpectrum software maps the generic module to the most optimal primitive memory cells, or Altera megafunction, for the target Altera technology.



For more information about inferring RAM and ROM megafunctions, including examples of VHDL and Verilog HDL code, see the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Inferring RAM

The LeonardoSpectrum software supports RAM inference for various device families. The restrictions for the LeonardoSpectrum software to successfully infer RAM in a design are listed below:

- The write process must be synchronous
- The read process can be asynchronous or synchronous depending on the target Altera architecture
- Resets on the memory are not supported

Table 9–3 shows a summary of the minimum memory sizes and minimum address widths for inferring RAM in various device families.

To disable RAM inference, set the `extract_ram` and `infer_ram` variables to "false." On the Tools menu, click **Variable Editor** to enter the value "false" when synthesizing in the user interface with the Advanced Flow tabs, or add the commands `set extract_ram false` and `set infer_ram false` to your synthesis script.

Table 9–3. Inferring RAM Summary

	Stratix Series and Cyclone Series	APEX Series, Excalibur and Mercury	FLEX 10KE and ACEX 1K
RAM primitive	`altsyncram`	`altdpram`	`altdpram`
Minimum RAM size	2 bits	64 bits	128 bits
Minimum address width	1 bit	4 bits	5 bits

Inferring ROM

You can implement ROM behavior in HDL source code with `CASE` statements or specify the ROM as a table. The LeonardoSpectrum software infers both synchronous and asynchronous ROM depending on the target Altera device. For example, memory for the Stratix series devices must be synchronous to be inferred.

To disable ROM inference, set the `extract_rom` variable to "false." To enter the value "false" when synthesizing in the user interface with the **Advanced Flow** tabs, on the Tools menu, click **Variable Editor**, or add the commands `set extract_rom false` to your synthesis script.

Inferring Multipliers and DSP Functions

Some Altera devices include dedicated DSP blocks optimized for DSP applications. The following Altera megafunctions are used with DSP block modes:

- `lpm_mult`
- `altmult_accum`
- `altmult_add`

You can instantiate these megafunctions in the design or have the LeonardoSpectrum software infer the appropriate megafunction by recognizing a multiplier, multiplier-accumulator (MAC), or multiplier-adder in the design. The Quartus II software maps the functions to the DSP blocks in the device during place-and-route.



For more information about inferring multipliers and DSP functions, including examples of VHDL and Verilog HDL code, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of *The Quartus II Handbook*.

Simple Multipliers

The `lpm_mult` megafunction implements the DSP block in the simple multiplier mode. The following functionality is supported in this mode:

- The DSP block includes registers for the input and output stages, and an intermediate pipeline stage
- Signed and unsigned arithmetic is supported

Multiplier Accumulators

The `altmult_accum` megafunction implements the DSP block in the multiply-accumulator mode. The following functionality is supported in this mode:

- The DSP block includes registers for the input and output stages, and an intermediate pipeline stage
- The output registers are required for the accumulator
- The input and pipeline registers are optional
- Signed and unsigned arithmetic is supported



If the design requires input registers to be used as shift registers, use the black-boxing method to instantiate the `altmult_accum` megafunction.

Multiplier Adders

The LeonardoSpectrum software can infer multiplier adders and map them to either the two-multiplier adder mode or the four-multiplier adder mode of the DSP blocks. The LeonardoSpectrum software maps the HDL code to the correct `altmult_add` function.

The following functionality is supported in these modes:

- The DSP block includes registers for the input and output stages and an intermediate pipeline stage
- Signed and unsigned arithmetic is supported, but support for the Verilog HDL "signed" construct is limited

Controlling DSP Block Inference

In devices that include dedicated DSP blocks, multipliers, multiply-accumulators, and multiply-adders can be implemented either in DSP blocks or in logic. You can control this implementation through attribute settings in the LeonardoSpectrum software.

As shown in Table 9–4, attribute settings in the LeonardoSpectrum software control the implementation of the multipliers in DSP blocks or logic at the signal block (or module), and project level.

Table 9–4. Attribute Settings for DSP Blocks in the LeonardoSpectrum Software *Note (1)*

Level	Attribute Name	Value	Description
Global	extract_mac *(2)*	TRUE	All multipliers in the project mapped to DSP blocks.
		FALSE	All multipliers in the project mapped to logic.
Module	extract_mac *(3)*	TRUE	Multipliers inside the specified module mapped to DSP blocks.
		FALSE	Multipliers inside the specified module mapped to logic.
Signal	dedicated_mult	ON	LPM inferred and multipliers implemented in DSP block.
		OFF	LPM inferred, but multipliers implemented in logic by the Quartus II software.
		LCELL	LPM not inferred, and multipliers implemented in logic by the LeonardoSpectrum software.
		AUTO	LPM inferred, but the Quartus II software automatically maps the multipliers to either logic or DSP blocks based on the Quartus II software place-and-route.

Notes to Table 9–4:

(1) The extract_mac attribute takes precedence over the dedicated_mult attribute.
(2) For devices with DSP blocks, the extract_mac attribute is set to "true" by default for the entire project.
(3) For devices with DSP blocks, the extract_mac attribute is set to "true" by default for all modules.

Global Attribute

You can set the global attribute extract_mac to control the implementation of multipliers in DSP blocks for the entire project. You can set this attribute using the script interface. The script command is:

```
set extract_mac <value>
```

Module Level Attributes

You can control the implementation of multipliers inside a module or component by setting attributes in the Verilog HDL source code. The attribute used is extract_mac. Setting this attribute for a module affects only the multipliers inside that module. The command is:

```
//synthesis attribute <module name> extract_mac <value>
```

The Verilog HDL and VHDL codes samples shown in Examples 9–1 and 9–2 show how to use the `extract_mac` attribute.

Example 9–1. Using Module Level Attributes in Verilog HDL Code

```
module mult_add ( dataa, datab, datac, datad, result);
//synthesis attribute mult_add extract_mac FALSE
// Port Declaration
input [15:0] dataa;
input [15:0] datab;
input [15:0] datac;
input [15:0] datad;

output [32:0] result;

// Wire Declaration
wire [31:0] mult0_result;
wire [31:0] mult1_result;

// Implementation
// Each of these can go into one of the 4 mults in a
// DSP block
assign mult0_result = dataa * `signed datab;
//synthesis attribute mult0_result preserve_signal TRUE

assign mult1_result = datac * datad;

// This adder can go into the one-level adder in a DSP
// block
assign result = (mult0_result + mult1_result);

endmodule
```

Example 9–2. Using Module Level Attributes in VHDL Code

```
library ieee ;
USE ieee.std_logic_1164.all;

USE ieee.std_logic_arith.all;

entity mult_acc is
   generic (size : integer := 4) ;
   port (
      a: in std_logic_vector (size-1 downto 0) ;
      b: in std_logic_vector (size-1 downto 0) ;
      clk : in std_logic;
        accum_out: inout std_logic_vector (2*size downto 0)
   ) ;
   attribute extract_mac : boolean;
   attribute extract_mac of mult_acc : entity is FALSE;
end mult_acc;

architecture synthesis of mult_acc is
    signal a_int, b_int : signed (size-1 downto 0);
    signal pdt_int : signed (2*size-1 downto 0);
    signal adder_out : signed (2*size downto 0);

begin
    a_int <= signed (a);
    b_int <= signed (b);
    pdt_int <= a_int * b_int;
    adder_out <= pdt_int + signed(accum_out);
    process (clk)
    begin
       if (clk'event and clk = '1') then
          accum_out <= std_logic_vector (adder_out);
       end if;
    end process;
end synthesis ;
```

Signal Level Attributes

You can control the implementation of individual `lpm_mult` multipliers by using the `dedicated_mult` attribute as shown below:

```
//synthesis attribute <signal_name> dedicated_mult <value>
```



The `dedicated_mult` attribute is only applicable to signals or wires; it is not applicable to registers.

Table 9–5 shows the supported values for the `dedicated_mult` attribute.

Table 9–5. Values for the dedicated_mult Attribute

Value	Description
ON	LPM inferred and multipliers implemented in DSP block.
OFF	LPM inferred and multipliers synthesized, implemented in logic, and optimized by the Quartus II software. *(1)*
LCELL	LPM not inferred and multipliers synthesized, implemented in logic, and optimized by the LeonardoSpectrum software. *(1)*
AUTO	LPM inferred but the Quartus II software maps the multipliers automatically to either the DSP block or logic based on resource availability.

Note to Table 9–5:

(1) Although both dedicated_mult=OFF and dedicated_mult=LCELLS result in logic implementations, the optimized results in these two cases may differ.



Some signals for which the `dedicated_mult` attribute is set may get synthesized away by the LeonardoSpectrum software due to design optimization. In such cases, if you want to force the implementation, the signal is preserved from being synthesized away by setting the `preserve_signal` attribute to "true."

The `extract_mac` attribute must be set to "false" for the module or project level when using the `dedicated_mult` attribute.

Examples 9–3 and 9–4 are samples of Verilog HDL and VHDL codes, respectively, using the `dedicated_mult` attribute.

Example 9–3. Signal Attributes for Controlling DSP Block Inference in Verilog HDL Code

```
module  mult (AX, AY, BX, BY, m, n, o, p);
input [7:0] AX, AY, BX, BY;
output [15:0] m, n, o, p;
wire [15:0] m_i = AX * AY; // synthesis attribute m_i dedicated_mult ON
// synthesis attribute m_i preserve_signal TRUE
//Note that the preserve_signal attribute prevents
// signal m_i from getting synthesized away
wire [15:0] n_i = BX * BY; // synthesis attribute n_i dedicated_mult OFF
wire [15:0] o_i = AX * BY; // synthesis attribute o_i dedicated_mult AUTO
wire [15:0] p_i = BX * AY; // synthesis attribute p_i dedicated_mult LCELL
// since n_i , o_i , p_i  signals are not preserved,
// they may be synthesized away based on the design
assign m = m_i;
assign n = n_i;
assign o = o_i;
assign p = p_i;
endmodule
```

Example 9–4. Signal Attributes for Controlling DSP Block Inference in VHDL Code

```
library ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_signed.all;

ENTITY mult is
PORT( AX,AY,BX,BY: IN
    std_logic_vector (17 DOWNTO 0);
m,n,o,p: OUT
    std_logic_vector (35 DOWNTO 0));
attribute dedicated_mult: string;
attribute preserve_signal : boolean
END mult;
ARCHITECTURE struct of mult is

signal m_i, n_i, o_i, p_i : unsigned (35 downto 0);
attribute dedicated_mult of m_i:signal is "ON";
attribute dedicated_mult of n_i:signal is "OFF";
attribute dedicated_mult of o_i:signal is "AUTO";
attribute dedicated_mult of p_i:signal is "LCELL";

begin

m_i <= unsigned (AX) * unsigned (AY);
n_i <= unsigned (BX) * unsigned (BY);
o_i <= unsigned (AX) * unsigned (BY);
p_i <= unsigned (BX) * unsigned (AY);

m <= std_logic_vector(m_i);
n <= std_logic_vector(n_i);
o <= std_logic_vector(o_i);
p <= std_logic_vector(p_i);
end struct;
```

Guidelines for Using DSP Blocks

In addition to the guidelines mentioned earlier in this section, use the following guidelines while designing with DSP blocks in the LeonardoSpectrum software:

- To access all the control signals for the DSP block, such as `sign A`, `sign B`, and `dynamic addnsub`, use the black-boxing technique.
- While performing signed operations, ensure that the specified data width of the output port matches the data width of the expected result. Otherwise, the sign bit may be lost or data may be incorrect because the sign is not extended.
  For example, if the data widths of input `A` and `B` are `width_a` and `width_b`, respectively, then the maximum data width of the result can be (`width_a` + `width_b` +2) for the four-multipliers adder mode. Thus, the data width of the output port should be less than or equal to (`width_a` + `width_b` +2).
- While using the accumulator, the data width of the output port should be equal to or greater than (`width_a` + `width_b`). The maximum width of the accumulator can be (`width_a` + `width_b` + 16). Accumulators wider than this are implemented in logic.
- If the design uses more multipliers than are available in a particular device, you may get a no fit error in the Quartus II software. In such cases, use the attribute settings in the LeonardoSpectrum software to control the mapping of multipliers in your design to DSP blocks or logic.

Block-Based Design with the Quartus II Software

The incremental compilation and LogicLock™ block-based design flows enable users to design, optimize, and lock down a design one section at a time. You can independently create and implement each logic module into a hierarchical or team-based design. With this method, you can preserve the performance of each module during system integration and have more control over placement of your design. To maximize the benefits of the incremental compilation or LogicLock design methodology in the Quartus II software, you can partition a new design into a hierarchy of netlist files during synthesis in the LeonardoSpectrum software.

The LeonardoSpectrum software allows you to create different netlist files for different sections of a design hierarchy. Different netlist files mean that each section is independent of the others. When synthesizing the entire project, only portions of a design that have been updated have to be re-synthesized when you compile the design. You can make changes, optimize, and re-synthesize your section of a design without affecting other sections.

For more information about incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about the LogicLock feature, refer to the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook.*

Hierarchy and Design Considerations

You must plan your design's structure and partitioning carefully to use incremental compilation and LogicLock features effectively. Optimal hierarchical design practices include partitioning the blocks at functional boundaries, registering the boundaries of each block, minimizing the I/O between each block, separating timing-critical blocks, and keeping the critical path within one hierarchical block.

For more recommendations for hierarchical design partitioning, refer to the *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook.*

To ensure the proper functioning of the synthesis tool, you can apply the LogicLock option in the LeonardoSpectrum software only to modules, entities, or netlist files. In addition, each module or entity should have its own design file. If two different modules are in the same design file but are defined as being part of different regions, it is difficult to maintain incremental synthesis since both regions would have to be recompiled when you change one of the modules or entities.

If you use boundary tri-states in a lower-level block, the LeonardoSpectrum software pushes (or "bubbles") the tri-states through the hierarchy to the top-level to take advantage of the tri-state drivers on the output pins of the Altera device. Because bubbling tri-states requires optimizing through hierarchies, lower-level tri-states are not supported with a block-level design methodology. You should use tri-state drivers only at the external output pins of the device and in the top-level block in the hierarchy.

If the hierarchy is flattened during synthesis, logic is optimized across boundaries, preventing you from making LogicLock assignments to the flattened blocks. Altera recommends preserving the hierarchy when compiling the design. In the **Optimize** command of your script, use the **Hierarchy Preserve** command or in the user interface select **Preserve** in the **Hierarchy** section on the **Optimize** Flow tab.

If you are compiling your design with a script, you can use an alternative method for preventing optimization across boundaries. In this case, use the **Auto** hierarchy setting and set the `auto_dissolve` attribute to false on the instances or views that you want to preserve (that is, the modules with LogicLock assignments) using the following syntax:

```
set_attribute -name auto_dissolve -value false
    .work.<block1>.INTERFACE
```

This alternative method flattens your design according to the `auto_dissolve` limits, but does not optimize across boundaries where you apply the attribute as described.



For more details on LeonardoSpectrum attributes and hierarchy levels, refer to the LeonardoSpectrum documentation in the Help menu.

Creating a Design with Multiple EDIF Files

The first stage of a hierarchical design flow is to generate multiple EDIF files, enabling you to take advantage of the incremental compilation flows in the Quartus II software. If the whole design is in one EDIF file, changes in one block affect other blocks because of possible node name changes. You can generate multiple EDIF files either by using the LogicLock option in the LeonardoSpectrum software, or by manually black boxing each block that you want to be part of a LogicLock region.

Once you have created multiple EDIF files using one of these methods, you must create the appropriate Quartus II project(s) to place-and-route the design.

Generating Multiple EDIF Files Using the LogicLock Option

This section describes how to generate multiple EDIF files using the LogicLock option in the LeonardoSpectrum software. When synthesizing a top-level design that includes LogicLock regions, use the following general steps:

1. Read in the Verilog HDL or VHDL source files.
2. Add LogicLock constraints.
3. Optimize and write output netlist files, or choose **Run** Flow.

To set the correct constraints and compile the design, use the following steps in the LeonardoSpectrum software:

1. Switch to the **Advanced** Flow tab instead of the **Quick Setup** tab (Tools menu).
2. Set the target technology and speed grade for the device on the **Technology** Flow tab.
3. Open the input source files on the **Input** Flow tab.
4. Click **Read** on the **Input** Flow tab to read the source files but not begin optimization.
5. Select the **Module** Power tab located at the bottom of the **Constraints** Flow tab.
6. Click on a module to be placed in a LogicLock region in the **Modules** section.
7. Turn on the **LogicLock** option.
8. Type the desired LogicLock region name in the text field under the **LogicLock** option.
9. Click **Apply**.
10. Repeat steps 6-9 for any other modules that you want to place in LogicLock regions.



In some cases, you are prompted to save your LogicLock and other non-global constraints in a Constraints File (**.ctr**) when you click anywhere off the **Constraints** Flow tab. The default name is <*project name*>**.ctr**. This file is added to your **Input** file list, and must be manually included later if you recreate the project.

The command written into the LeonardoSpectrum Information or Transcript Window is the Tcl command that gets written into the CTR file. The format of the "path" for the module specified in the command should be work.<*module*>.**INTERFACE**. To ensure that you don't see an optimized version of the module, do not perform a **Run Flow** on the **Quick Setup** tab prior to setting LogicLock constraints. Always use the **Read** command, as described in step 4.

11. Continue making any other settings as required on the **Constraints** tab.

12. Select **Preserve** in the **Hierarchy** section on the **Optimize** tab to ensure that the hierarchy names are not flattened during optimization.

13. Continue making any other settings as required on the **Optimize** tab.

14. Run your synthesis flow using each Flow tab, or click **Run Flow**.

Synthesis creates an EDIF file for each module that has a LogicLock assignment in the **Constraints** Flow tab. You can now use these files with the incremental compilation flows in the Quartus II software.



You might occasionally see multiple EDIF files and LogicLock commands for the same module. An "unfolded" version of a module is created when you instantiate a module more than once and the boundary conditions of the instances are different. For example, if you apply a constant to one instance of the block, it might be optimized to eliminate unneeded logic. In this case, the LeonardoSpectrum software must create a separate module for each instantiation (unfolding). If this unfolding occurs, you see more than one EDIF file, and each EDIF file has a LogicLock assignment to the same LogicLock region. When you import the EDIF files to the Quartus II software, the EDIF files created from the module are placed in different LogicLock regions. Any optimizations performed in the Quartus II software using the LogicLock methodology must be performed separately for each EDIF netlist.

Creating a Quartus II Project for Multiple EDIF Files Including LogicLock Regions

The LeonardoSpectrum software creates Tcl files that provide the Quartus II software with the appropriate LogicLock assignments, creating a region for each EDIF file along with the information to set up a Quartus II project.

The Tcl file contains the commands shown in Example 9–5 for each LogicLock region. This example is for module `taps` where the name `taps_region` was typed as the LogicLock region name in the **Constraints** Flow tab in the LeonardoSpectrum software.

Example 9–5. Tcl File for Module Taps with taps_region as LogicLock Region Name

```
project add_assignment {taps} {taps_region} {} {}
    {LL_AUTO_SIZE} {ON}
project add_assignment {taps} {taps_region} {} {}
    {LL_STATE} {FLOATING}
project add_assignment {taps} {taps_region} {} {}
    {LL_MEMBER_OF} {taps_region}
```

These commands create a LogicLock region with Auto-Size and Floating-Origin properties. This flexible LogicLock region allows the Quartus II Compiler to select the size and location of the region.



For more information about Tcl commands, refer to the *TCL Scripting* chapter in volume 2 of the *Quartus II Handbook*.

You can use the following methods to import the EDIF file and corresponding Tcl file into the Quartus II software:

- Use the Tcl file that is created for each EDIF file by the LeonardoSpectrum software. This method allows you to generate multiple Quartus II projects, one for each block in the design. Each designer in the project can optimize their block separately in the Quartus II software and preserve their results. Altera recommends this method for bottom-up incremental and hierarchical design methodologies because it allows each block in the design to be treated separately. Each block can be brought into one top-level project with the import function.

  or

- Use the <*top-level project*>**.tcl** file that contains the assignments for all blocks in the project. This method allows the top-level designer to import all the blocks into one Quartus II project. You can optimize all modules in the project at once in a top-down design flow. If additional optimization is required for individual blocks, each designer can use their EDIF file to create a separate project at that time. You would then have to add new assignments to the top-level project using the import function.

In both methods, you can use the following steps to create the Quartus II project, import the appropriate LogicLock assignments, and compile the design:

1. Place the EDIF and Tcl files in the same directory.
2. On the View menu, point to Utility Windows and click **Tcl Console** to open the Quartus II **Tcl Console**.

3. Type `source <path>/<project name>.tcl` ↵.

4. To open the new completed project, on the File menu, click **Open Project**. Browse to and select the project name, and click **Open**.



For more information about importing design using incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about importing LogicLock assignments, see the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*.

Generating Multiple EDIF Files Using Black Boxes

This section describes how to manually generate multiple EDIF files using the black-boxing technique. The manual flow, described below, was supported in older versions of the LeonardoSpectrum software. The manual flow is discussed here because some designers want more control over the project for each submodule.

To create multiple EDIF files in the LeonardoSpectrum software, create a separate project for each module and top-level design that you want to maintain as a separate EDIF file. Implement black-box instantiations of lower-level modules in your top-level project.

When synthesizing the projects for the lower-level modules and the top-level design, use the following general guidelines.

For lower-level modules:

- Turn off **Map IO Registers** for the target technology on the **Technology** Flow tab.
- Read the HDL files for the modules. Modules may include black-box instantiations of lower-level modules that are also maintained as separate EDIF files.
- Add constraints.
- Turn off **Add I/O Pads** on the **Optimize** Flow tab.

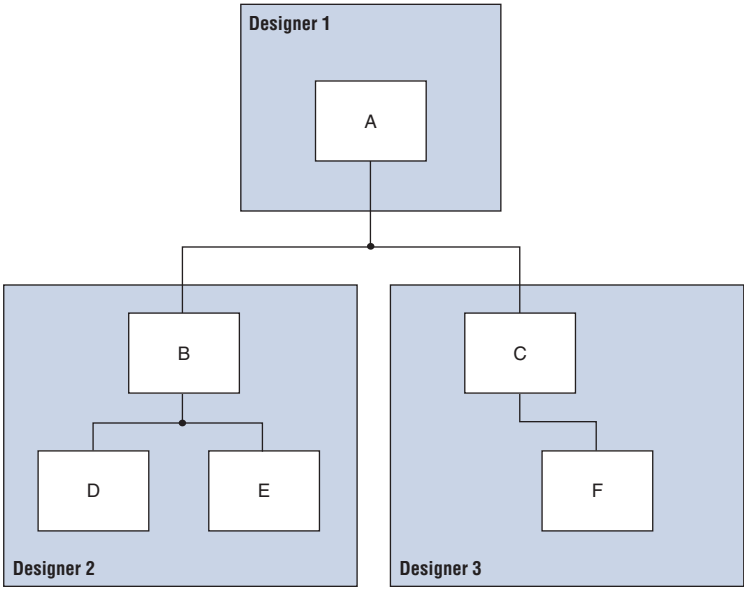
For the top-level design:

- Turn on **Map IO Registers** if you want to implement input and/or output registers in the IOEs for the target technology on the **Technology** Flow tab.
- Read the HDL files for the top-level design.
  - Black-box lower-level modules in the top-level design
- Add constraints (clock settings should be made at this time).

The following sections describe examples of black-box modules in a block-based and team-based design flow.

In Figure 9–3, the top-level design A is assigned to one engineer (designer 1), while two-engineers work on the lower levels of the design. Designer 2 works on B and its submodules D and E, while designer 3 works on C and its submodule F.

Figure 9–3. Block-Based and Team-Based Design Example



One netlist is created for the top-level module A, another netlist is created for B and its submodules D and E, while another netlist is created for C and its submodule F. To create multiple EDIF files, perform the following steps:

1. Generate an EDIF file for module C. Use **C.v** and **F.v** as the source files.
2. Generate an EDIF file for module B. Use **B.v**, **D.v**, and **E.v** as the source files.
3. Generate a top-level EDIF file **A.v** for module A. Ensure that your black-box modules B and C were optimized separately in steps 1 and 2.

Black Boxing in Verilog HDL

Any design block that is not defined in the project, or included in the list of files to be read for a project, is treated as a black box by the software. In Verilog HDL, you must also provide an empty module declaration for the module that you plan to treat as a black box.

Example 9–6 shows an example of the **A.v** top-level file. If any of your lower-level files also contain a black-boxed lower-level file in the next level of hierarchy, follow the same procedure.

Example 9–6. Verilog HDL Top-Level File Black-Boxing Example

```
module A (data_in,clk,e,ld,data_out);
    input data_in, clk, e, ld;
    output [15:0] data_out;

    reg [15:0] cnt_out;
    reg [15:0] reg_a_out;

    B U1 ( .data_in (data_in),.clk (clk), .e(e), .ld (ld),
           .data_out(cnt_out) );

    C U2 ( .d(cnt_out), .clk (clk), .e(e), .q (reg_out));
    // Any other code in A.v goes here.

endmodule

// Empty Module Declarations of Sub-Blocks B and C follow here.
// These module declarations (including ports) are required for blackboxing.

module B (data_in,e,ld,data_out );
    input data_in, clk, e, ld;
    output [15:0] data_out;
endmodule

module C (d,clk,e,q );
    input d, clk, e;
    output [15:0] q;
endmodule
```



Previous versions of the LeonardoSpectrum software required an attribute statement `//exemplar attribute U1 NOOPT TRUE`, which instructs the software to treat the instance `U1` as a black box. This attribute is no longer required, although it is still supported in the software.

Black Boxing in VHDL

Any design block that is not defined in the project, or included in the list of files to be read for a project, is treated as a black box by the software. In VHDL, you need a component declaration for the black box which is normal for any other block in the design.

Example 9–7 shows an example of the **A.vhd** top-level file. If any of your lower-level files also contain a black-boxed lower-level file in the next level of hierarchy, follow the same procedure.

Example 9–7. VHDL Top-Level File Black-Boxing Example

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY A IS
PORT ( data_in : IN INTEGER RANGE 0 TO 15;
    clk : IN STD_LOGIC;
    e : IN STD_LOGIC;
    ld : IN STD_LOGIC;
    data_out : OUT INTEGER RANGE 0 TO 15
);
END A;

ARCHITECTURE a_arch OF A IS

COMPONENT B PORT(
    data_in : IN INTEGER RANGE 0 TO 15;
    clk : IN STD_LOGIC;
    e : IN STD_LOGIC;
    ld : IN STD_LOGIC;
    data_out : OUT INTEGER RANGE 0 TO 15
);
END COMPONENT;

COMPONENT C PORT(
    d : IN INTEGER RANGE 0 TO 15;
    clk : IN STD_LOGIC;
    e : IN STD_LOGIC;
    q : OUT INTEGER RANGE 0 TO 15
);
END COMPONENT;

-- Other component declarations in A.vhd go here

signal cnt_out : INTEGER RANGE 0 TO 15;
signal reg_a_out : INTEGER RANGE 0 TO 15;
BEGIN
CNT : C
PORT MAP (
    data_in => data_in,
    clk => clk,
    e => e,
    ld => ld,
    data_out => cnt_out
);

REG_A : D
PORT MAP (
    d => cnt_out,
    clk => clk,
    e => e,
    q => reg_a_out
);

-- Any other code in A.vhd goes here

END a_arch;
```



Previous versions of the LeonardoSpectrum software required the attribute statement `noopt of C: component is TRUE`, which instructed the software to treat the component C as a black box. This attribute is no longer required, although it is still supported in the software.

After you have completed the steps outlined in this section, you have a different EDIF netlist file for each block of code. You can now use these files for incremental compilation flows in the Quartus II software.

Creating a Quartus II Project for Multiple EDIF Files

The LeonardoSpectrum software creates a Tcl file for each EDIF file, which provides the Quartus II software with the information to set up a project.

As in the previous section, there are two different methods for bringing each EDIF and corresponding Tcl file into the Quartus II software:

- Use the Tcl file that is created for each EDIF file by the LeonardoSpectrum software. This method generates multiple Quartus II projects, one for each block in the design. Each designer in the project can optimize their block separately in the Quartus II software and preserve their results. Designers should create a LogicLock region for each block; the top-level designer should then import all the blocks and assignments into the top-level project. Altera recommends this method for bottom-up incremental and hierarchical design methodology because it allows each block in the design to be treated separately; each block can be imported into one top-level project.

  or

- Use the <*top-level project*>**.tcl** file that contains the information to set up the top-level project. This method allows the top-level designer to create LogicLock regions for each block and bring all the blocks into one Quartus II project. Designers can optimize all modules in the project at once in a top-down design flow. If additional optimization is required for individual blocks, each designer can take their EDIF file and create a separate Quartus II project at that time. New assignments would then have to be added to the top-level project manually or through the import function.



For more information about importing designs using incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about importing LogicLock regions, refer to the *LogicLock Design Methodology* chapter in the volume 2 of the *Quartus II Handbook*.

In both methods, you can use the following steps to create the Quartus II project and compile the design:

1. Place the EDIF and Tcl files in the same directory.
2. On the View menu, point to Utility Windows and click **Tcl Console**. The Quartus II **Tcl Console** is shown.
3. At a Tcl prompt, type `source` <*path*>/<*project name*>`.tcl` ↵.
4. On the File menu, click **Open Project**. In the **New Project** window, browse to and select the project name. Click **Open**.
5. To create LogicLock assignments, on the Assignments menu, click **LogicLock Regions Window**.
6. On the Processing menu, click **Start Compilation**.

Incremental Synthesis Flow

If you make changes to one or more submodules, you can manually create new projects in the LeonardoSpectrum software to generate a new EDIF netlist file when there are changes to the source files. Alternatively, you can use incremental synthesis to generate a new netlist for the changed submodule(s). To perform incremental synthesis in the LeonardoSpectrum software, use the script described in this section to reoptimize and generate a new EDIF netlist for only the affected modules using the LeonardoSpectrum top-level project. This method applies only when you are using the **LogicLock** option in the LeonardoSpectrum software.

Modifications Required for the LogicLock_Incremental.tcl Script File

There are three sets of entries in the file that must be modified before beginning incremental synthesis. The variables in the Tcl file are surrounded by angle brackets (< >).

1. Add the list of source files that are included in the project. You can enter the full path to the file or just the file name if the files are located in the working directory.

2. Indicate which modules in the design have changed. These modules are the EDIF files that are regenerated by the LeonardoSpectrum software. These modules contain a LogicLock assignment in the original compilation.

   Obtain the LeonardoSpectrum software path for each module by looking at the CTR file that contains the LogicLock assignments from the original project. Each LogicLock assignment is applied to a particular module in the design.

3. Enter the target device family using the appropriate device keyword. The device keyword is written into the **Transcript** or **Information** window when you select a target Technology and click **Load Library** or **Apply** on the **Technology Flow** tab in the graphical user interface.

Example 9–8 shows the **LogicLock_Incremental.tcl** file for the incremental synthesis flow. You must modify the Tcl file before you can use it for your project.

Example 9–8. LogicLock_Interface.tcl Script File for Incremental Synthesis

```
##############################################
#### LogicLock Incremental Synthesis Flow ####
##############################################

## You must indicate which modules have changed (based on the source files
## that have changed) and provide the complete path to each module

## You must also specify the list of design files and the target Altera
## technology being used

# Read the design source files.
read <list of design files separated by spaces (such as block1.v block2.v)>

# Get the list of modified modules in bottom-up "depth first search" order
# where the lower-level blocks are listed first (these should be modules
# that had LogicLock assignments and separate EDIF netlist files in the
# first pass and had their source code modified)

set list_of_modified_modules {.work.<block2>.INTERFACE .work.<block1>.INTERFACE}

foreach module $list_of_modified_modules {
    set err_rc [regexp {\.(.*)\.(.*)\.(.*)} $module unused lib module_name arch]
    present_design $module

    # Run optimization, preserving hierarchy. You must specify a technology.
    optimize -ta <technology> -hierarchy preserve

    # Ensure that the lower-level module is not optimized again when
    # optimizing higher-level modules.
    dont_touch $module
}

foreach module $list_of_modified_modules {
    set err_rc [regexp {\.(.*)\.(.*)\.(.*)} $module unused lib module_name arch]
    present_design $module
    undont_touch $module
    auto_write $module_name.edf
    # Ensure that the lower-level module is not written out in the EDIF file
    # of the higher-level module.
    noopt $module
}
```

Running the Tcl Script File in LeonardoSpectrum

Once you have modified the Tcl script, as described in the "Modifications Required for the LogicLock_Incremental.tcl Script File" on page 9–31, you can compile your design using the script.

You can run the script in batch mode at the command line prompt using the following command:

```
spectrum -file <Tcl_file> ↵
```

To run the script from the interface, on the File menu, click **Run Script**, then browse to your Tcl file and click **Open**.

The LogicLock incremental design flow uses module-based design to help you preserve performance of modules and have control over placement. By tagging the modules that require separate EDIF files, you can make multiple EDIF files for use with the Quartus II software from a single LeonardoSpectrum software project.

Conclusion

Advanced synthesis is an important part of the design flow. Taking advantage of the Mentor Graphics LeonardoSpectrum software and the Quartus II design flow allows you to control how your design files are prepared for the Quartus II place-and-route process, as well as to improve performance and optimize a design for use with Altera devices. The methodologies outlined in this chapter can help optimize a design to achieve performance goals and save design time.

Referenced Documents

This chapter references the following documents:

- *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbookk*
- *LeonardoSpectrum Installation Guide* and the *LeonardoSpectrum User's Manual*.
- *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*
- *Mentor Graphics Precision RTL Synthesis Support* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *TCL Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 9–6 shows the revision history of this chapter.

Table 9–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Stratix III device added. • Reorganized "Referenced Documents" on page 9–34.	Version updated for 2007.
May 2007 v7.1.0	• Updated LeonardoSpectrum software version to 2006b • Added "Referenced Documents" on page 9–34.	—
November 2006 v6.1.0	Added document revision history to chapter.	—
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.	—
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 10 was formerly chapter 11 in version 5.0.	—
May 2005 v5.0.0	Chapter 11 was formerly chapter 9 in version 4.2.	—
December 2004 v2.1.0	• Chapter 10 was formerly Chapter 11 in version 4.1. • Updated information. • New functionality in Quartus II software version 4.2. • Updated tables and figures.	—
June 2004 v2.0.0	• Updates to tables, and figures. • New functionality for Quartus II software version 4.1.	—
Feb. 2004 v1.0.0	Initial release.	—

10. Mentor Graphics Precision RTL Synthesis Support

QII51011-7.2.0

Introduction

As programmable logic device (PLD) designs become more complex and require increased performance, advanced synthesis has become an important part of the design flow. When integrated into the Quartus II® design flow, Mentor Graphics® Precision RTL Synthesis can be used to improve performance results for Altera® devices.

This chapter documents support for the Mentor Graphics Precision RTL Synthesis software in the Quartus II software design flow, as well as key design methodologies and techniques for improving your results for Altera devices.

The topics discussed in this chapter include:

- "Design Flow" on page 10–2
- "Creating a Project and Compiling the Design" on page 10–6
- "Mapping the Precision Synthesis Design" on page 10–7
- "Synthesizing the Design and Evaluating the Results" on page 10–13
- "Exporting Designs to the Quartus II Software Using NativeLink Integration" on page 10–14
- "Megafunctions and Architecture-Specific Features" on page 10–23
- "Incremental Compilation and Block-Based Design" on page 10–32

This chapter assumes that you have installed and licensed the Precision RTL Synthesis software and the Quartus II software.



To obtain and license the Precision RTL Synthesis software, refer to the Mentor Graphics web site at **www.mentor.com**. To install and run the Precision RTL Synthesis software and to set up your work environment, refer to the *Precision RTL Synthesis User's Manual* in the Precision Manuals Bookcase in the Help menu.

Device Family Support

The following list shows the Altera device families supported by the Mentor Graphics Precision RTL Synthesis software version 2007a when used with the Quartus II software version 7.2:

- Arria™ GX
- Cyclone® III
- Stratix® III
- Stratix II, Stratix II GX, Hardcopy® II
- Stratix, Stratix GX, HardCopy Stratix
- Cyclone II
- Cyclone
- MAX® 3000A, MAX 7000, MAX 7000AE, MAX 7000B, MAX 7000E, MAX 7000S, MAX II
- ACEX® 1K
- APEX™ 20K, APEX 20KC, APEX 20KE, APEX II
- FLEX® 10K, FLEX 10KA, FLEX 10KB, FLEX 10KE, FLEX 6K

The Precision software also supports the following legacy devices that are supported in the Quartus II software (as applicable to the specific license requested at the support section of **www.altera.com)**.

- Excalibur™ ARM®
- Mercury™

In addition, the Precision software supports the following legacy devices that are supported in the Altera MAX+PLUS II software only:

- MAX 9000
- FLEX 8000

Design Flow

The basic steps in a Quartus II design flow using the Precision RTL Synthesis software includes:

1. Create Verilog HDL or VHDL design files in the Quartus II design software, the Precision RTL Synthesis software, or with a text editor.
2. Create a project in the Precision RTL Synthesis software that contains the HDL files for your design, select your target device, and set global constraints.
3. Compile the project in the Precision RTL Synthesis software.
4. Add specific timing constraints, optimization attributes, and compiler directives to optimize the design during synthesis.



For best results, Mentor Graphics recommends specifying constraints that are as close as possible to actual operating requirements. Properly setting clock and I/O constraints, assigning clock domains, and indicating false and multicycle paths guide the synthesis algorithms more accurately toward a suitable solution in the shortest synthesis time.

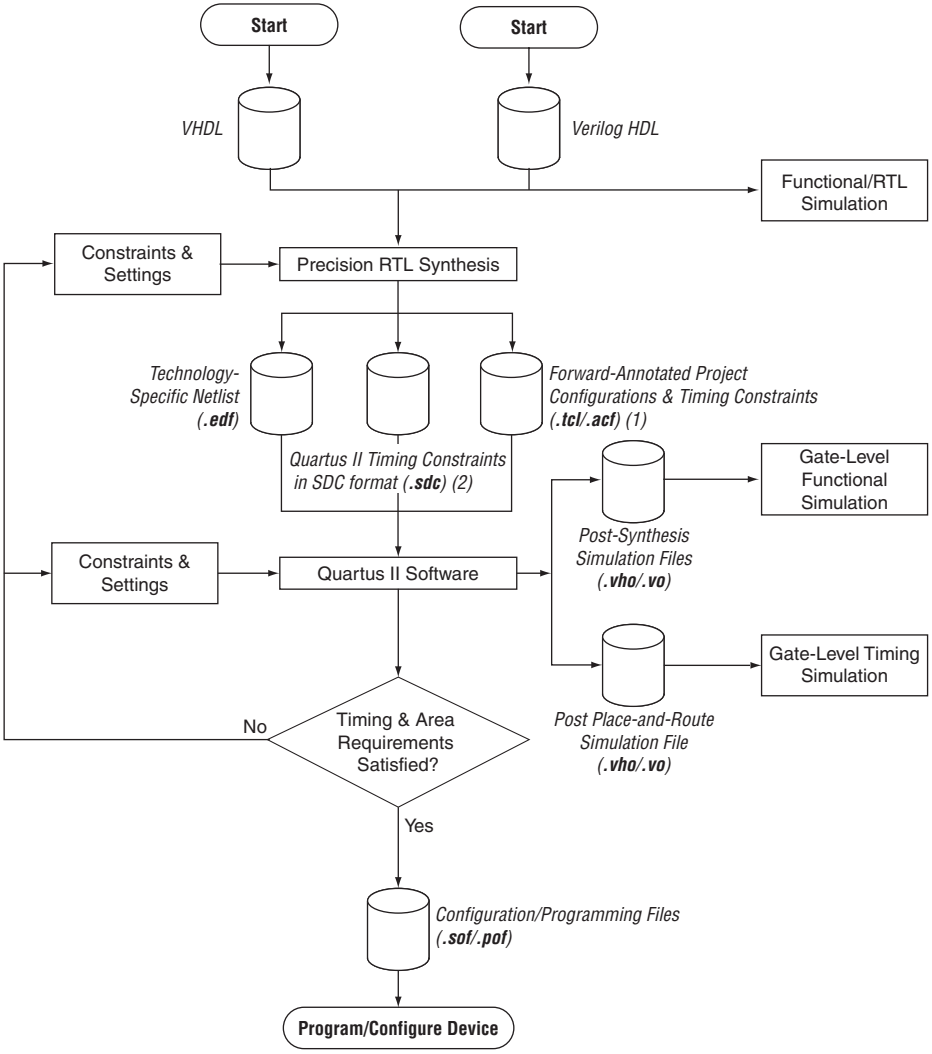
5. Synthesize the project in the Precision RTL Synthesis software. With the design analysis capabilities and cross-probing of Precision RTL Synthesis software, you can identify and improve circuit area and performance issues using pre-layout timing estimates.

6. Create a Quartus II project and import the technology-specific EDIF (**.edf**) netlist, the Quartus II design constraints file (**.sdc**) in Synopsys Design Constraints (SDC) format (TimeQuest constraints if a Stratix III, Arria GX, or Cyclone III device is selected), and the tool command language (**.tcl**) file generated by the Precision RTL Synthesis software into the Quartus II software for placement and routing, and for performance evaluation using actual post-layout timing data.

7. After obtaining place-and-route results that meet your needs, configure or program the Altera device.

These steps are described in detail throughout this chapter. Figure 10–1 shows the Quartus II design flow using Precision RTL Synthesis as described in the above steps.

Figure 10–1. Design Flow Using the Precision RTL Synthesis Software and Quartus II Software



Notes to Figure 10–1:

(1) If a device other than Stratix III, Arria GX, or Cyclone III is selected, two **.tcl** files are generated. One file acts as a Quartus II Project Configuration file, while the other file contains timing constraints for the Classic Timing Analyzer. If a Stratix III, Arria GX, or Cyclone III device is selected, only the Quartus II Project Configuration **.tcl** file is generated.

(2) This file is generated automatically only if a Stratix III, Arria GX, or Cyclone III device is selected.

If your area or timing requirements are not met, you can change the constraints and resynthesize the design in the Precision RTL Synthesis software, or you can change constraints to optimize the design during place-and-route in the Quartus II software. Repeat the process until the area and timing requirements are met (Figure 10–1).

You can use other options and techniques in the Quartus II software to meet area and timing requirements. One such option is the **WYSIWYG Primitive Resynthesis** option, which can perform optimizations on your EDIF netlist in the Quartus II software.



For information about netlist optimizations, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*. For more recommendations about how to optimize your design, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

While simulation and analysis can be performed at various points in the design process, final timing analysis should be performed after placement and routing is complete.

During the synthesis process, the Precision RTL Synthesis software produces several intermediate and output files. Table 10–1 lists these files with a short description of each file type.

Table 10–1. Precision RTL Synthesis Software Intermediate and Output Files

File Extension(s)	File Description
.psp	Precision RTL Synthesis Software Project File
.xdb	Mentor Graphics Design Database File
.rep *(1)*	Synthesis Area and Timing Report File
.edf	Technology-specific netlist in electronic design interchange format (EDIF)
.acf/**.tcl** *(2)*	Forward-annotated constraints file containing constraints and assignments
.sdc *(3)*	Quartus II timing constraints file in Synopsys Design Constraints format

Notes to Table 10–1:

(1) The timing report file includes performance estimates that are based on pre-place-and-route information. Use the f_{MAX} reported by the Quartus II software after place-and-route for accurate post-place-and-route timing information. The area report file includes post-synthesis device resource utilization statistics that may differ from the resource usage after place-and-route due to black-boxes or further optimizations performed during placement and routing. Use the device utilization reported by the Quartus II software after place-and-route for final resource utilization results. See "Synthesizing the Design and Evaluating the Results" on page 10–13 for details.

(2) An Assignment and Configuration File (**.acf**) file is created only for ACEX 1K, FLEX 10K, FLEX 10KA, FLEX 6000, FLEX 8000, MAX 7000, MAX 9000, and MAX 3000 devices. The Assignment and Configuration File is generated for backward compatibility with the MAX+PLUS® II software. Two **.tcl** files are generated, depending on the device selected: <*project name*>**.tcl** and <*project name*>**_pnr_constraints.tcl**. The <*project name*>**.tcl** file is generated regardless of device selected. It acts as the Quartus II Project Configuration file and is used to make basic project assignments and to create and compile a Quartus II project for your EDIF netlist. If an Stratix III, Arria GX, or Cyclone III device is selected, this file contains the command required to use the TimeQuest Timing Analyzer instead of the Classic Timing Analyzer. The <*project name*>**_pnr_constraints.tcl** file is generated automatically only if a device other than Stratix III, Arria GX, or Cyclone III is selected. It contains timing constraints called by <*project name*>**.tcl** to perform placement and routing.

(3) This file is generated automatically only if a Stratix III, Arria GX, or Cyclone III device is selected, and has the naming convention <*project name*>**_pnr_constraints.sdc**

Creating a Project and Compiling the Design

After creating your design files, create a project in the Precision RTL Synthesis software that contains the basic settings for compiling the design.

Creating a Project

Set up your design files as follows:

1. In the Precision RTL Synthesis software, click the **New Project** icon in the Design Bar on the left side of the GUI.
2. Set the **Project Name** and the **Project Folder**. The implementation name of the design corresponds to this project name.

3. Add input files to the project with the **Add Input Files** icon in the Design Bar. Precision RTL Synthesis software automatically detects the top-level module/entity of the design. It uses the top-level module/entity to name the current implementation directory, logs, reports, and netlist files.

4. In the Design Bar, click the **Setup Design** icon.

5. To specify a target device family, expand the Altera entry, and choose the target device and speed grade.

6. If desired, set a global design frequency and/or default input and output delays. This constrains all clock paths and all I/O pins in your design. Modify the settings for individual paths or pins that do not require such a setting.

To generate additional netlist files (for example, an HDL netlist for simulation), on the Tools menu, point to **Set Options** and **Output** and select the desired output format. The Precision RTL Synthesis software generates a separate file for each selected type of file: EDIF, Verilog HDL, and VHDL.

Compiling the Design

To compile the design into a technology-independent implementation, on the Design Bar, click the **Compile** icon.

Mapping the Precision Synthesis Design

In the next steps, you set constraints and map the design to technology-specific cells. The Precision RTL Synthesis software maps the design by default to the fastest possible implementation that meets your timing constraints. To accomplish this, you must specify timing requirements for the automatically determined clock sources. With this information, the Precision RTL Synthesis software performs static timing analysis to determine the location of the critical timing paths. The Precision RTL Synthesis software achieves the best results for your design when you set as many realistic constraints as possible. Be sure to set constraints for timing, mapping, false paths, multicycle paths, and others that control the structure of the implemented design.

Mentor Graphics recommends creating a Synopsys Design Constraint file (**.sdc**) and adding this file to the **Constraint Files** section of the Project Files list. You can create this file with a text editor, by issuing command line constraint parameters, or using the Precision RTL Synthesis software to generate one automatically for you on the first synthesis run. To create a constraint file with the user interface, set constraints on design objects (such as clocks, design blocks, or pins) in the Design Hierarchy browser.

By default, the Precision RTL Synthesis software saves all timing constraints and attributes in two files: **precision_rtl.sdc** and **precision_tech.sdc**. The **precision_rtl.sdc** file contains constraints set on the RTL-level database (after compilation) and the **precision_tech.sdc** file contains constraints set on the gate-level database (after synthesis) located in the current implementation directory.

You can also enter constraints at the command line. After adding constraints at the command line, update the **.sdc** file with the **update constraint file** command.



You can add constraints that change infrequently directly to the HDL source files with HDL attributes or pragmas.



For more details and examples, refer to the *Attributes* chapter in the *Precision Synthesis Reference Manual* in the Precision Manual Bookcase in the Help menu.

Setting Timing Constraints

Timing constraints, based on the industry-standard Synopsys Design Constraint file format, help the Precision RTL Synthesis software to deliver optimal results. Missing timing constraints can result in incomplete timing analysis and may prevent timing errors from being detected. Precision RTL Synthesis software provides constraint analysis prior to synthesis to ensure that designs are fully and accurately constrained. If a device other than Stratix III, Arria GX, or Cyclone III is selected, all timing constraints are forward-annotated to the Quartus II software using Tcl scripts for the Quartus II Classic Timing Analyzer. If a Stratix III, Arria GX, or Cyclone III device is selected, **<***project name***>_pnr_constraints.sdc** is generated that contains timing constraints in SDC format. So, instead of using the Classic Timing Analyzer, the Quartus II software uses the TimeQuest Timing Analyzer for timing analysis.



Because the Synopsys Design Constraint file format requires that timing constraints must be set relative to defined clocks, you must specify your clock constraints before applying any other timing constraints.

You also can use multicycle path and false path assignments to relax requirements or exclude nodes from timing requirements. Doing so can improve area utilization and allow the software optimizations to focus on the most critical parts of the design.



For details about the syntax of Synopsys Design Constraint commands, refer to the *Precision RTL Synthesis Users Manual* and the *Precision Synthesis Reference Manual* available in the Precision Manual Bookcase in the Help menu.

Setting Mapping Constraints

Mapping constraints affect how your design is mapped into the target Altera device. You can set mapping constraints in the user interface, in HDL code, or with the **set_attribute** command in the constraint file.

Assigning Pin Numbers and I/O Settings

The Precision RTL Synthesis software supports assigning device pin numbers, I/O standards, drive strengths, and slew-rate settings to top-level ports of the design. You can set these timing constraints with the **set_attribute** command, the GUI, or by specifying synthesis attributes in your HDL code. These constraints are written into the Tcl file that is read by the Quartus II software during place-and-route and do not affect synthesis.

You can use the **set_attribute** command in the Synopsys Design Constraint file to specify pin number constraints, I/O standards, drive strengths, and slow slew-rate settings. Table 10–2 outlines the format to use for entries in the Synopsys Design Constraint file.

Table 10–2. Constraint File Settings

Constraint	Entry Format for Synopsys Design Constraint File
Pin number	`set_attribute -name PIN_NUMBER -value "<pin number>" -port <port name>`
I/O standard	`set_attribute -name IOSTANDARD -value "<I/O Standard>" -port <port name>`
Drive strength	`set_attribute -name DRIVE -value "<drive strength in mA>" -port <port name>`
Slew rate	`set_attribute -name SLEW -value "TRUE | FALSE" -port <port name>`

You can also specify these options in the GUI. To specify a pin number or other I/O setting in the Precision RTL Synthesis GUI, follow these steps:

1. After compiling the design, expand the **Ports** entry in the Design Hierarchy Browser.
2. Under **Ports,** expand the **Inputs** or **Outputs** entry.



You also can assign I/O settings by right-clicking the pin in the Schematic Viewer.

3. Right-click the desired pin name and select the **Set Input Constraints** option under **Inputs** or **Set Output Constraints** option under **Outputs**.

4. Enter the desired pin number on the Altera device in the **Pin Number** box (**Port Constraints** dialog box).

5. Select the I/O standard from the **IO_STANDARD** list.

6. For output pins, you can also select a drive strength setting and slew rate setting using the **DRIVE** and **SLOWSLEW** lists.

You also can use synthesis attributes or pragmas in your HDL code to make these assignments. Example 10–1 and Example 10–2 show code samples that makes a pin assignment in your HDL code.

Example 10–1. Verilog HDL Pin Assignment

```
//pragma attribute clk pin_number P10;
```

Example 10–2. VHDL Pin Assignment

```
attribute pin_number : string
attribute pin_number of clk : signal is "P10";
```

You can use the same syntax to assign the I/O standard using the attribute `IOSTANDARD`, drive strength using the attribute `DRIVE`, and slew rate using the attribute `SLEW`.



For more details about attributes and how to set these attributes in your HDL code, refer to the *Precision Synthesis Reference Manual*. To access this manual, in the Synplify software, click **Help** and select **Open Manuals Bookcase**.

Assigning I/O Registers

The Precision RTL Synthesis software performs timing-driven I/O register mapping by default. It moves registers into an I/O element (IOE) when it does not negatively impact the register-to-register performance of your design, based on the timing constraints.

You can force a register to the device's IOE using the Complex I/O constraint. This option does not apply if you turn off **I/O pad insertion.** Refer to "Disabling I/O Pad Insertion" on page 10–11 for more information.

To force an I/O register into the device's IOE using the GUI, follow these steps:

1. After compiling the design, expand the **Ports** entry in the Design Hierarchy browser.
2. Under **Ports**, expand the **Inputs** or **Outputs** entry, as desired.
3. Under **Inputs** or **Outputs**, right-click the desired pin name, point to **Map Input Register to IO** or **Map Output Register to IO** for input or output respectively, and click **True**.



You also can make the assignment by right-clicking on the pin in the Schematic Viewer.

For the Stratix and Cyclone series, and MAX II device families, the Precision RTL Synthesis software can move an internal register to an I/O register without any restrictions on design hierarchy.

For more mature devices, the Precision RTL Synthesis software can move an internal register to an I/O register only when the register exists in the top level of the hierarchy. If the register is buried in the hierarchy, you must flatten the hierarchy so that the buried registers are moved to the top level of the design.

Disabling I/O Pad Insertion

The Precision RTL Synthesis software assigns I/O pad atoms (device primitives used to represent the I/O pins and I/O registers) to all ports in the top level of a design by default. In certain situations, you may not want the software to add I/O pads to all I/O pins in the design. The Quartus II software can compile a design without I/O pads; however, including I/O pads provides the Precision RTL Synthesis software with the most information about the top-level pins in the design.

Preventing the Precision RTL Synthesis Software from Adding I/O Pads

If you are compiling a subdesign as a separate project, I/O pins cannot be primary inputs or outputs of the device and therefore should not have an I/O pad associated with them. To prevent the Precision RTL Synthesis software from adding I/O pads, perform the following steps:

1. On the Tools menu, click **Set Options**.
2. On the **Optimization** page of the **Options** dialog box, turn off **Add IO Pads**, then click **Apply**.

This procedure adds the following command to the project file:

```
setup_design -addio=false
```

Preventing the Precision RTL Synthesis Software from Adding an I/O Pad on an Individual Pin

To prevent I/O pad insertion on an individual pin when you are using a black box, such as Double Data Rate (DDR) or a Phase-Locked Loop (PLL), at the external ports of the design, follow these steps:

1. After compiling the design, in the Design Hierarchy browser, expand the **Ports** entry by clicking the + icon.
2. Under **Ports**, expand the **Inputs** or **Outputs** entry.
3. Under **Inputs** or **Outputs**, right-click the desired pin name and click **Set Input Constraints**.
4. In the **Port Constraints** dialog box for the selected pin name, turn off **Insert Pad**.



You also can make the assignment by right-clicking on the pin in the Schematic Viewer or by attaching the `nopad` attribute to the port in the HDL source code.

Controlling Fan-Out on Data Nets

Fan-out is defined as the number of nodes driven by an instance or top-level port. High fan-out nets can have significant delays which can result in an unroutable net. On a critical path, high fan-out nets can cause larger delay in a single net segment which can result in the timing constraints not being met. To prevent this behavior, each device family has a global fan-out value set in the Precision RTL Synthesis software

library. In addition, the Quartus II software automatically routes high fan-out signals on global routing lines in the Altera device whenever possible.

To eliminate routability and timing issues associated with high fan-out nets, the Precision RTL Synthesis software also allows you to override the library default value on a global or individual net basis. You can override the library value by setting a `max_fanout` attribute on the net.

Synthesizing the Design and Evaluating the Results

To synthesize the design for the target device, click on the **Synthesize** icon in the Precision RTL Synthesis Design Bar. During synthesis, the Precision RTL Synthesis software optimizes the compiled design, then writes out netlists and reports to the implementation subdirectory of your working directory after the implementation is saved, using the naming convention:

<*project name*>`_impl_`<*number*>

After synthesis is complete, you can evaluate the results in terms of area and timing. The *Precision RTL Synthesis User's Manual* on the Mentor Graphics web site describes different results that can be evaluated in the software.

There are several schematic viewers available in the Precision RTL Synthesis software: RTL schematic, Technology-mapped schematic, and Critical Path schematic. These analysis tools allow you to quickly and easily isolate the source of timing or area issues, and to make additional constraint or code changes to optimize the design.

Obtaining Accurate Logic Utilization and Timing Analysis Reports

Historically, designers have relied on post-synthesis logic utilization and timing reports to determine how much logic their design requires, how big a device they need, and how fast the design will run. However, today's FPGA devices provide a wide variety of advanced features in addition to basic registers and look-up tables (LUTs). The Quartus II software has advanced algorithms to take advantage of these features, as well as optimization techniques to both increase performance and reduce the amount of logic required for a given design. In addition, designs may contain black boxes and functions that take advantage of specific device features. Because of these advances, synthesis tool reports provide post-synthesis area and timing estimates, but the place-and-route software should be used to obtain final logic utilization and timing reports.

Exporting Designs to the Quartus II Software Using NativeLink Integration

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools, which allows you to run other EDA design entry/synthesis, simulation, and timing analysis tools automatically from within the Quartus II software.

After a design is synthesized in the Precision RTL Synthesis software, the technology-mapped design is written to the current implementation directory as an EDIF netlist file, along with a Quartus II Project Configuration File and a place-and-route constraints file. You can use the Project Configuration script, <*project name*>**.tcl**, to create and compile a Quartus II project for your EDIF netlist. This script makes basic project assignments, such as assigning the target device specified in the Precision RTL Synthesis software. For the Quartus II Timing Analyzer, the Project Configuration script calls the place-and-route constraints script, <*project name*>**_pnr_constraints.tcl**, to make your timing constraints. If you select an Arria GX, Stratix III, or Cyclone III device, the constraints are written in SDC format to the **<***project name***>_pnr_constraints.sdc** file by default and is used by the Fitter and the TimeQuest Timing Analyzer in the Quartus II software.

If you want to use the Quartus II TimeQuest Timing Analyzer, use the following Precision command before compilation:

```
setup_design -timequest_sdc
```

With this command, a file named <*project name*>**_pnr_constraints.sdc** is generated after the synthesize command. You can also use the following command to generate the SDC constraint file but not set up the entire project to use TimeQuest:

```
report_constraints -timequest
```

The `report_constraints - timequest` command also generates the file <*project_name*>**_pnr_constraints.sdc**

Running the Quartus II Software from within the Precision RTL Software

Precision RTL Synthesis software also has a built-in place-and-route environment that allows you to run the Quartus II Fitter and view the results in the Precision RTL Synthesis GUI. This feature is useful when performing an initial compilation of your design to view post-place-and-route timing and device utilization results, but not all the advanced Quartus II options that control the compilation process are available.

After you specify an Altera device as the target, set the options for the Quartus II software. On the Tools menu, click **Set Options**. On the **Integrated Place and Route** page (under **Quartus II Modular**), specify the path to the Quartus II executables in the **Path to Quartus II installation tree** box.

To automate the place-and-route process, click the **Run Quartus II** icon in the **Quartus II Modular** window of the Precision RTL Synthesis Toolbar. The Quartus II software uses the current implementation directory as the Quartus II project directory and runs a full compilation in the background (that is, no user interface appears).

Two primary Precision RTL Synthesis software commands control the place-and-route process. Place-and-route options are set by the **setup_place_and_route** command. The process is started with the **place_and_route** command.

Precision RTL Synthesis software versions 2004a and later support using individual Quartus II executables, such as analysis and synthesis (**quartus_map**), Fitter (**quartus_fit**), and the Classic Timing Analyzer (**quartus_tan**) or the TimeQuest Timing Analyzer (**quartus_sta**) (only for software version 2006a and later), for improved runtime and memory utilization during place and route. This flow is referred to as the **Quartus II Modular** flow option in Precision RTL Synthesis software and is compatible with Quartus II software versions beginning with version 4.0. By default, the Precision RTL Synthesis software generates this Quartus II Project Configuration File (Tcl file) for Arria GX, Stratix series, MAX II, and Cyclone series device families. When you use this flow, all timing constraints that you set during synthesis are exported to the Quartus II place-and-route constraints file <*project name*>**_pnr_constraints.tcl**, or **<***project name***>_pnr_constraints.sdc**, depending on which Quartus II timing analyzer the Precision RTL Synthesis software is targeting.

For other device families, the Precision RTL Synthesis software uses the **Quartus II** flow option, which enables the Quartus II compilation flow that existed in Precision RTL Synthesis software versions earlier than 2004a. The Quartus II Project Configuration File (Tcl file) is written when using the **Quartus II** flow option that includes supported timing constraints that you specified during synthesis. This Tcl file is compatible with all versions of the Quartus II software; however, the format and timing constraints do not take full advantage of the features in the Quartus II software introduced with version 4.0.

To force the use of a particular flow when it is not the default for a certain device family, use the following command to set up the integrated place-and-route flow:

```
setup_place_and_route -flow "<Altera Place-and-Route flow>"
```

Depending on the device family, you can use one of the following flow options in the command above:

- `Quartus II Modular`
- `Quartus II`
- `MAX+PLUS II`

For example, for the Stratix II or MAX II device families (which were not supported in Quartus II software versions earlier than 4.0), you can use only the **Quartus II Modular** flow. For the Stratix device family, you can use either the **Quartus II Modular** or **Quartus II** flows. The FLEX 8000 device family, which is not supported in the Quartus II software, is supported only by the **MAX+PLUS II** flow.

After the design is compiled in the Quartus II software from within the Precision RTL Synthesis software, you can invoke the Quartus II GUI manually and then open the project using the generated Quartus II project file. You can view reports, run analysis tools, specify options, and run the various processing flows available in the Quartus II software.

Running the Quartus II Software Manually Using the Precision RTL Synthesis-Generated Tcl Script

You can use the Quartus II software separately from the Precision RTL Synthesis software. To run the Tcl script generated by the Precision RTL Synthesis software to set up your project and start a full compilation, perform the following steps:

1. Ensure the EDIF, Tcl files, and SDC file (if using the TimeQuest Timing Analyzer) are located in the same directory (by default, the files should be located in the implementation directory).
2. In the Quartus II software, on the View menu, point to Utility Windows and click **Tcl Console**.
3. At the Tcl Console command prompt, type the command:

   `source` *<path>*`/`*<project name>*`.tcl` ↵
4. On the File menu, click **Open Project**. Browse to the project name, and click **Open**.
5. Compile the project in the Quartus II software.

Using Quartus II Software to Launch the Precision RTL Synthesis Software

Using NativeLink integration, you can set up the Quartus II software to run the Precision RTL Synthesis software. This feature allows you to use the Precision RTL Synthesis software to synthesize a design as part of a normal compilation.



For detailed information about using NativeLink integration with the Precision RTL Synthesis software, go to *Specifying EDA Tool Settings* in the Quartus II Help index.

Passing Constraints to the Quartus II Software

The place-and-route constraints script forward-annotates timing constraints that you made in the Precision RTL Synthesis software. This integration allows you to enter these constraints once in the Precision RTL Synthesis software, and then pass them automatically to the Quartus II software.



All of the constraints you set in the Precision software are mapped to the Quartus II software. For same constraint you set in the Precision software, there may be a different command mapped to the Quartus II software, depending on whether you are using the TimeQuest Timing Analyzer or the Classic Timing Analyzer.



Refer to the introductory text in the section Exporting Designs to the Quartus II Software Using NativeLink Integration on page 10–14 for information on how to ensure Precision targets the TimeQuest Timing Analyzer.

The following constraints are translated by the Precision RTL Synthesis software and are applicable to the Classic Timing Analyzer and the TimeQuest Timing Analyzer:

- `create_clock`
- `set_input_delay`
- `set_output_delay`
- `set_max_delay`
- `set_min_delay`
- `set_false_path`
- `set_multicycle_path`

create_clock

You can specify a clock in the Precision RTL Synthesis software, as shown in Example 10–3.

Example 10–3. Specifying a Clock using create_clock

```
create_clock -name <clock_name> -period <period in ns> -waveform {<edge_list>}
-domain <ClockDomain> <pin>
```

The period is specified in units of nanoseconds (ns). If no clock domain is specified, the clock belongs to a default clock domain `main`. All clocks in the same clock domain are treated as synchronous (related) clocks. If no <*clock_name*> is provided, the default name `virtual_default` is used. The <*edge_list*> sets the rise and fall edges of the clock signal over an entire clock period. The first value in the list is a rising transition, typically the first rising transition after time zero. The waveform can contain any even number of alternating edges, and the edges listed should alternate between rising and falling. The position of any edge can be equal to or greater than zero but must be equal to or less than the clock period.

If `-waveform` <*edge_list*> is not specified, and `-period` <*period_value*> is specified, the default waveform has a rising edge of 0.0 and a falling edge of <*period_value*>/2.

The Precision RTL Synthesis software passes the clock definitions to the Quartus II software with the `create_base_clock` command in the place-and-route constraints file for the Classic Timing Analyzer. For the TimeQuest Timing Analyzer, the clock constraint is mapped to the TimeQuest `create_clock` setting in the Quartus II software.

The following list describes some differences in the clock properties supported by the Precision RTL Synthesis software and the Quartus II software:

- The Quartus II software supports only clock waveforms with two edges in a clock cycle. If the Precision RTL Synthesis software finds a multi-edge clock, it issues an error message when you synthesize your design in Precision RTL Synthesis software. This applies to both the Quartus II TimeQuest Timing Analyzer and the Quartus II Classic Timing Analyzer.
- Clocks in the same clock `-domain` are annotated with the `create_relative_clock` command to create related clocks for the Quartus II Classic Timing Analyzer.
- The Quartus II Classic Timing Analyzer assumes the first clock edge to be at time 0.0. If the Precision RTL Synthesis software waveform has a first transition at a time different than time zero (0.0), the

Precision RTL Synthesis software creates a base clock without any target, then uses this to create a relative clock with an offset set to the first clock edge.

set_input_delay

This port-specific input delay constraint is specified in the Precision RTL Synthesis software, as shown in Example 10–4.

Example 10–4. Specifying set_input_delay

```
set_input_delay {<delay_value> <port_pin_list>} -clock <clock_name> -rise
-fall -add_delay
```

This constraint is mapped to the `set_input_delay` setting in the Quartus II software.

When the reference clock <*clock_name*> is not specified, all clocks are assumed to be the reference clocks for this assignment. The input pin name for the assignment can be an input pin name of a time group. The software can use the option `clock_fall` to specify delay relative to the falling edge of the clock.



Although the Precision RTL Synthesis software allows you to set input delays on pins inside the design, these constraints are not sent to the Quartus II software, and a message is displayed.

set_output_delay

This port-specific output delay constraint is specified in the Precision RTL Synthesis software, as shown in Example 10–5.

Example 10–5. Using the set_output_delay Constraint

```
set_output_delay {<delay_value> <port_pin_list>} -clock <clock_name> -rise -fall
-add_delay
```

This constraint is mapped to the `set_output_delay` setting in the Quartus II software.

When the reference clock <*clock_name*> is not specified, all clocks are assumed to be the reference clocks for this assignment. The output pin name for the assignment can be an output pin name of a time group.



Although the Precision RTL Synthesis software allows you to set output delays on pins inside the design, these constraints are not sent to the Quartus II software.

set_max_delay

The total delay for a point-to-point timing path constraint is specified in the Precision RTL Synthesis software, as shown in Example 10–6.

Example 10–6. Using the set_max_delay Constraint

```
set_max_delay -from {<from_node_list>} -to {<to_node_list>} <delay_value>
```

This command specifies that the maximum required delay for any start point in <*from_node_list*> to any endpoint in <*to_node_list*> must be less than <*delay_value*>. Typically, this command is used to override the default setup constraint for any path with a specific maximum time value for the path.

The node lists can contain a collection of clocks, registers, ports, pins, or cells. The `-from`, and `-to` parameters specify the source (start point), and the destination (endpoint) of the timing path respectively. The source list (<*from_node_list*>) cannot include output ports, and the destination list (<*to_node_list*>) cannot include input ports. If you include more than one node on a list, you must enclose the nodes in quotes or in '{ }' braces.

If you specify a clock in the source list, you must specify a clock in the destination list. Applying `set_max_delay` between clocks applies the exception from all registers or ports driven by the source clock to all registers or ports driven by the destination clock. Applying exceptions between clocks is more efficient than applying them for specific node to node, or node to clock paths. If you want to specify pin names in the list, the source must be a clock pin, and the destination must be any non-clock input pin to a register. Assignments from clock pins, or to and from cells, apply to all registers in the cell or for those driven by the clock pin.

set_min_delay

The minimum delay for a point-to-point timing path constraint is specified in the Precision RTL Synthesis software, as shown in Example 10–7.

Example 10–7. Using the set_min_delay Constraint

```
set_min_delay -from {<from_node_list>} -to {<to_node_list>} <delay_value>
```

This command specifies that the minimum required delay for any start point in <*from_node_list*> to any endpoint in <*to_node_list*> must be greater than <*delay_value*>. Typically, you use this command to override the default setup constraint for any path with a specific minimum time value for the path.

The node lists can contain a collection of clocks, registers, ports, pins, or cells. The -from, and -to specify the source (start point), and the destination (endpoint) of the timing path respectively. The source list (<*from_node_list*>) cannot include output ports, and the destination list (<*to_node_list*>) cannot include input ports. If you include more than one node to a list, you must enclose the nodes in quotes or in '{ }' braces.

If you specify a clock in the source list, you must specify a clock in the destination list. Applying `set_min_delay` between clocks applies the exception from all registers or ports driven by the source clock to all registers or ports driven by the destination clock. Applying exceptions between clocks is more efficient than applying them for specific node to node, or node to clock paths. If you want to specify pin names in the list, the source must be a clock pin, and the destination must be any non-clock input pin to a register. Assignments from clock pins, or to and from cells, apply to all registers in the cell or for those driven by the clock pin.

set_false_path

The false path constraint is specified in the Precision RTL Synthesis software, as shown in Example 10–8.

Example 10–8. Using the set_false_path Constraint

```
set_false_path -to <to_node_list> -from <from_node_list> -reset_path
```

The node lists can be a list of clocks, ports, instances, and pins. Multiple elements in the list can be represented using wildcards such as "*" and "?."

In place-and-route Tcl constraints file, this setting in the Precision RTL Synthesis software is mapped to a `set_timing_cut_assignment` setting for the Classic Timing Analyzer. For the TimeQuest Timing Analyzer, this constraint is mapped to the `set_false_path` setting.

The node lists for this assignment represents top-level ports and/or nets connected to instances (end points of timing assignments).

The Quartus II software supports `setup`, `hold`, `rise`, or `fall` options for this assignment only if you are using the TimeQuest Timing Analyzer.

The Quartus II Classic Timing Analyzer does not support false paths with the `through` path specification. Any setting in the Precision RTL Synthesis software with a `through` specification can be mapped to a setting in the Quartus II software only if you use the TimeQuest Timing Analyzer.

For the Classic Timing Analyzer, if you use the `from` or `to` option without using both options, the Precision RTL Synthesis command is converted to a Quartus II command using wildcards. Table 10–3 lists these `set_false_path` constraints in the Precision RTL Synthesis software and the Quartus II software equivalent when the Classic Timing Analyzer is used.

Table 10–3. set_false_path Constraints with the Classic Timing Analyzer

Precision RTL Synthesis Assignment	Quartus II Equivalent
set_false_path -from <*from_node_list*>	set_timing_cut_assignment -to {*} -from <*node_list*>
set_false_path -to <*to_node_list*>	set_timing_cut_assignmet -to <*node_list*> -from {*}

set_multicycle_path

This multi-cycle path constraint is specified in the Precision RTL Synthesis software, as shown in Example 10–9.

Example 10–9. Using the set_multicycle_path Constraint

```
set_multicycle_path <multiplier_value> [-start] [-end] -to <to_node_list> -from
<from_node_list> -reset_path
```

The node lists can contain clocks, ports, instances, and pins. Multiple elements in the list can be represented using wildcards such as "`*`" and "`?`." Paths without multicycle path definitions are identical to paths with multipliers of 1. To add one additional cycle to the datapath, use a multiplier value of 2. The option `start` is to indicate that source clock cycles should be considered for the multiplier. The option `end` is to indicate that destination clock cycles should be considered for the multiplier. The default is to reference the end clock.

In the place-and-route Tcl constraints file, this setting in the Precision RTL Synthesis software is mapped to a `set_multicycle_assignment` setting for the Classic Timing Analyzer. For TimeQuest Timing Analyzer, this constraint is mapped to the `set_multicycle_path` setting.

The node lists represent top-level ports and/or nets connected to instances (end points of timing assignments). The node lists can contain wildcards (such as '*'); the Quartus II software automatically expands all wildcards.

For the Classic Timing Analyzer, if you use the `from` or `to` option without using both options, the Precision RTL Synthesis command is converted to a Quartus II command using wildcards. Table 10–4 lists the `set_multicycle_path` constraints in the Precision RTL Synthesis software and the Quartus II software equivalent, when the Classic Timing Analyzer is used.

Table 10–4. set_multicycle_path Constraints for the Classic Timing Analyzer

Precision RTL Synthesis Assignment	Quartus II Equivalent
`set_multicycle_path -from` <*from_node_list*> <*value*>	`set_multicycle_assignment -to {*} -from` <*node_list*> <*value*>
`set_multicycle_path -to` <*to_node_list*> <*value*>	`set_multicycle_assignmet -to` <*node_list*> `-from {*}` <*value*>

The Quartus II software supports the `rise` or `fall` options on this assignment only if you use the TimeQuest Timing Analyzer.

The Quartus II Classic Timing Analyzer does not support multicycle path with a `through` path specification. Any setting in Precision RTL Synthesis software with a `-through` specification can be mapped to a setting in the Quartus II software only if you use the TimeQuest Timing Analyzer.

Megafunctions and Architecture-Specific Features

Altera provides parameterizable megafunctions including LPM, device-specific Altera megafunctions, intellectual property (IP) available as Altera MegaCore functions, and IP available through the Altera Megafunction Partners Program (AMPP℠). You can use megafunctions by instantiating them in your HDL code or inferring them from generic HDL code.



For more details about specific Altera megafunctions, refer to the Quartus II Help. For more information about IP functions, consult the appropriate IP documentation.

To instantiate a megafunction in your HDL code, you can use the MegaWizard® Plug-In Manager to parameterize the function or you can instantiate the function using the port and parameter definition. The

MegaWizard Plug-In Manager provides a graphical interface for customizing and parameterizing any available megafunction for the design. "Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager" on page 10–24 describes the MegaWizard flow with the Precision RTL Synthesis software.

The Precision RTL Synthesis software automatically recognizes certain types of HDL code and infers the appropriate megafunction when a megafunction will provide optimal results. The Precision RTL Synthesis software also provides options to control inference of certain types of megafunctions, as described in "Inferring Altera Megafunctions from HDL Code" on page 10–25.



For a detailed information about instantiating versus inferring megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. This chapter also provides details about using the MegaWizard Plug-In Manager in the Quartus II software and explains the files generated by the wizard. In addition, the chapter provides coding style recommendations and examples for inferring megafunctions in Altera devices.

Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager

When you use the MegaWizard Plug-In Manager to set up and parameterize a megafunction and to create a custom megafunction variation, the MegaWizard creates either a VHDL or Verilog HDL wrapper file.

Instantiating the MegaWizard Plug-In Manager-generated wrapper file is referred to as a black-box methodology because the megafunction is treated as a black box in the Precision RTL Synthesis software.



Beginning with Quartus II software version 7.1, there is an option in the MegaWizard Plug-In Manager to create a netlist for area and timing estimation instead of a wrapper file. This option is not currently supported with the Precision RTL Synthesis software, therefore you must use the megafunction wrapper file as described in this section.

Using MegaWizard Plug-In Manager-Generated Verilog HDL Files for Black-Box Megafunction Instantiation

The MegaWizard Plug-In Manager generates a Verilog HDL instantiation template file <*output file*>**_inst.v** and a hollow-body black-box module declaration <*output file*>**_bb.v** for use in your Precision RTL Synthesis design. The instantiation template file helps to instantiate the megafunction variation wrapper file, <*output file*>**.v**, in your top-level

design. Add the hollow-body black-box module declaration <*output file*>**_bb.v** to your Precision RTL Synthesis project to describe the port connections of the black box.

Including the megafunction variation wrapper file <*output file*>**.v** in your Precision RTL Synthesis project is optional, but you must add it to your Quartus II project along with your Precision RTL synthesis-generated EDIF netlist. Alternately, you can include the file in your Precision project and then right-click on the file in the input file list, and select **Properties**. In the **input file properties** dialog box, turn on **Exclude file from Compile Phase** and click **OK**. When this option is on, the Precision RTL Synthesis software does not compile this file and the tool makes a copy of the file in the appropriate directory so that the Quartus II software can compile the design during placement and routing.

Using MegaWizard Plug-In Manager-Generated VHDL Files for Black-Box Megafunction Instantiation

The MegaWizard Plug-In Manager generates a VHDL Component declaration file <*output file*>**.cmp** and a VHDL Instantiation template file <*output file*>**_inst.vhd** for use in your Precision RTL Synthesis design. These files can help you instantiate the megafunction variation wrapper file, <*output file*>.**vhd**, in your top-level design.

Including the megafunction variation wrapper file, <*output file*>.**vhd**, in your Precision RTL synthesis project is optional, but you must add it to your Quartus II project with your Precision RTL synthesis-generated EDIF netlist. Alternately, you can include the file in your Precision project and then right-click on the file in the input file list, and select **Properties**. In the **input file properties** dialog box, turn on **Exclude file from Compile Phase** and click **OK**. When this option is on, the Precision RTL Synthesis software does not compile this file and the tool makes a copy of the file in the appropriate directory so that the Quartus II software can compile the design during placement and routing.

Inferring Altera Megafunctions from HDL Code

The Precision RTL Synthesis software automatically recognizes certain types of HDL code and maps arithmetic and relational operators, and memory (RAM and ROM), to efficient technology-specific implementations. This allows for the use of technology-specific resources to implement these structures by inferring the appropriate Altera megafunction when a megafunction will provide optimal results. In some cases, the Precision RTL Synthesis software has options that you can use to disable or control inference.



For coding style recommendations and examples for inferring megafunctions in Altera devices, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*, and the *Precision Synthesis Style Guide* in the Precision RTL Synthesis Manuals Bookcase in the Help menu.

Multipliers

The Precision RTL Synthesis software detects multipliers in HDL code and maps them directly to device atoms to implement the multiplier in the appropriate type of logic. The Precision RTL Synthesis software also allows you to control the device resources that are used to implement individual multipliers, as described in the following section.

Controlling DSP Block Inference for Multipliers

By default, the Precision RTL Synthesis software uses DSP blocks available in the Stratix series of devices to implement multipliers. The default setting is **AUTO**, to allow Precision RTL Synthesis software the flexibility to choose between logic look-up tables (LUTs) and DSP blocks, depending on the size of the multiplier. You can use the Precision RTL Synthesis GUI or HDL attributes to direct the mapping to only logic elements or to only DSP blocks. The options for multiplier mapping in the Precision RTL Synthesis software are shown in Table 10–5.

Table 10–5. Options for dedicated_mult Parameter to Control Multiplier Implementation in Precision RTL Synthesis

Value	Description
ON	Use only DSP blocks to implement multipliers, regardless of the size of the multiplier.
OFF	Use only logic (LUTs) to implement multipliers.
AUTO	Use logic (LUTs) and DSP blocks to implement multipliers depending on the size of the multipliers.

Using the GUI

To set the **Use Dedicated Multiplier** option in the Precision RTL Synthesis GUI, perform the following steps:

1. Compile the design.
2. In the Design Hierarchy browser, right-click the operator for the desired multiplier and click **Use Dedicated Multiplier**.

Using Attributes

To control the implementation of a multiplier in your HDL code, use the `dedicated_mult` attribute with the appropriate value from Table 10–5 on page 10–26, as shown in Example 10–10 and Example 10–11.

Example 10–10. Setting the dedicated_mult Attribute in Verilog HDL

```
//synthesis attribute <signal name> dedicated_mult <value>
```

Example 10–11. Setting the dedicated_mult Attribute in VHDL

```
ATTRIBUTE dedicated_mult: STRING;
ATTRIBUTE dedicated_mult OF <signal name>: SIGNAL IS <value>;
```

The `dedicated_mult` attribute can be applied to signals and wires; it does not work when applied to a register. This attribute can be applied only to simple multiplier code, such as `a = b * c`.

Some signals for which the `dedicated_mult` attribute is set may be synthesized away by the Precision RTL Synthesis software because of design optimization. In such cases, if you want to force the implementation, you should preserve the signal by setting the `preserve_signal` attribute to `TRUE`, as shown in Example 10–12 and Example 10–13.

Example 10–12. Setting the preserve_signal Attribute in Verilog HDL

```
//synthesis attribute <signal name> preserve_signal TRUE
```

Example 10–13. Setting the preserve_signal Attribute in VHDL

```
ATTRIBUTE preserve_signal: BOOLEAN;
ATTRIBUTE preserve_signal OF <signal name>: SIGNAL IS TRUE;
```

Example 10–14 and Example 10–15 are examples in Verilog HDL and VHDL of using the dedicated_mult attribute to implement the given multiplier in regular logic in the Quartus II software.

Example 10–14. Verilog HDL Multiplier Implemented in Logic

```
module unsigned_mult (result, a, b);
    output [15:0] result;
    input [7:0] a;
    input [7:0] b;
    assign result = a * b; //synthesis attribute result dedicated_mult OFF
endmodule
```

Example 10–15. VHDL Multiplier Implemented in Logic

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY unsigned_mult IS
    PORT(
       a: IN std_logic_vector (7 DOWNTO 0);
       b: IN std_logic_vector (7 DOWNTO 0);
       result: OUT std_logic_vector (15 DOWNTO 0));
ATTRIBUTE dedicated_mult: STRING;
END unsigned_mult;

ARCHITECTURE rtl OF unsigned_mult IS
    SIGNAL a_int, b_int: UNSIGNED (7 downto 0);
    SIGNAL pdt_int:  UNSIGNED (15 downto 0);
ATTRIBUTE dedicated_mult OF pdt_int: SIGNAL IS "OFF";
BEGIN
    a_int <= UNSIGNED (a);
    b_int <= UNSIGNED (b);
    pdt_int <= a_int * b_int;
    result <= std_logic_vector(pdt_int);
END rtl;
```

Multiplier-Accumulators and Multiplier-Adders

The Precision RTL Synthesis software detects multiply-accumulators or multiply-adders in HDL code and infers an altmult_accum or altmult_add megafunction so that the logic can be placed in DSP blocks, or maps directly to device atoms to implement the multiplier in the appropriate type of logic.



The Precision RTL Synthesis software supports inference for these functions only if the target device family has dedicated DSP blocks.

The Precision RTL Synthesis software also allows you to control the device resources used to implement multiply-accumulators or multiply-adders in your project or in a particular module. Refer to the "Controlling DSP Block Inference" on page 10–29 section for more information.



For more information about DSP blocks in Altera devices, refer to the appropriate Altera device family handbook and device-specific documentation. For details about which functions a given DSP block can implement, refer to the DSP Solutions Center on the Altera web site at **www.altera.com**.



For more information about inferring Multiply-Accumulator and Multiply-Adder megafunctions in HDL code, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*, and the *Precision Synthesis Style Guide* in the Precision RTL Synthesis Manuals Bookcase in the Help menu.

Controlling DSP Block Inference

By default, the Precision RTL Synthesis software infers the altmult_add or altmult_accum megafunction as appropriate for your design. These megafunctions allow the Quartus II software the flexibility to choose regular logic or DSP blocks depending on the device utilization and the size of the function.

You can use the `extract_mac` attribute to prevent the inference of an altmult_add or altmult_accum megafunction in a certain module or entity. The options for this attribute are shown in Table 10–6.

Table 10–6. Options for extract_mac Attribute Controlling DSP Implementation

Value	Description
TRUE	The altmult_add or altmult_accum megafunction is inferred
FALSE	The altmult_add or altmult_accum megafunction is not inferred

To control inference, use the `extract_mac` attribute with the appropriate value from Table 10–6 on page 10–29 in your HDL code, as shown in Example 10–16 and Example 10–17.

Example 10–16. Setting the extract_mac Attribute in Verilog HDL

```
//synthesis attribute <module name> extract_mac <value>
```

Example 10–17. Setting the extract_mac Attribute in VHDL

```
ATTRIBUTE extract_mac: BOOLEAN;
ATTRIBUTE extract_mac OF <entity name>: ENTITY IS <value>;
```

To control the implementation of the multiplier portion of a multiply-accumulator or multiply-adder, you must use the `dedicated_mult` attribute as described in "Controlling DSP Block Inference" on page 10–29 (see this section for syntax details).

Example 10–18 and Example 10–19 use the `extract_mac`, `dedicated_mult`, and `preserve_signal` attributes (in Verilog HDL and VHDL) to implement the given DSP function in logic in the Quartus II software.

Example 10–18. Using extract_mac, dedicated_mult and preserve_signal in Verilog HDL

```
module unsig_altmult_accum1 (dataout, dataa, datab, clk, aclr, clken);
    input [7:0] dataa, datab;
    input clk, aclr, clken;
    output     [31:0] dataout;

    reg        [31:0] dataout;
    wire       [15:0] multa;
    wire       [31:0] adder_out;

    assign multa = dataa * datab;

    //synthesis attribute multa preserve_signal TRUE
    //synthesis attribute multa dedicated_mult OFF
    assign adder_out = multa + dataout;

    always @ (posedge clk or posedge aclr)
    begin
        if (aclr)
        dataout <= 0;
    else if (clken)
        dataout <= adder_out;
    end

    //synthesis attribute unsig_altmult_accum1 extract_mac FALSE
endmodule
```

Example 10–19. Using extract_mac, dedicated_mult, and preserve_signal in VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;

ENTITY signedmult_add IS
    PORT(
        a, b, c, d: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        result: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
    );
    ATTRIBUTE preserve_signal: BOOLEAN;
    ATTRIBUTE dedicated_mult: STRING;
    ATTRIBUTE extract_mac: BOOLEAN;
    ATTRIBUTE extract_mac OF signedmult_add: ENTITY IS FALSE;
END signedmult_add;

ARCHITECTURE rtl OF signedmult_add IS
    SIGNAL a_int, b_int, c_int, d_int : signed (7 DOWNTO 0);
    SIGNAL pdt_int, pdt2_int : signed (15 DOWNTO 0);
    SIGNAL result_int: signed (15 DOWNTO 0);
```

```
    ATTRIBUTE preserve_signal OF pdt_int: SIGNAL IS TRUE;
    ATTRIBUTE dedicated_mult OF pdt_int: SIGNAL IS "OFF";
    ATTRIBUTE preserve_signal OF pdt2_int: SIGNAL IS TRUE;
    ATTRIBUTE dedicated_mult OF pdt2_int: SIGNAL IS "OFF";

BEGIN
    a_int <= signed (a);
    b_int <= signed (b);
    c_int <= signed (c);
    d_int <= signed (d);
    pdt_int <= a_int * b_int;
    pdt2_int <= c_int * d_int;
    result_int <= pdt_int + pdt2_int;
    result <= STD_LOGIC_VECTOR(result_int);
END rtl;
```

RAM and ROM

The Precision RTL Synthesis software detects memory structures in HDL code and converts them to an operator that infers an `altsyncram` or `lpm_ram_dp` megafunction, depending on the device family. The software then places these functions in memory blocks.

The software supports inference for these functions only if the target device family has dedicated memory blocks.



For more information about inferring RAM and ROM megafunctions in HDL code, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook,* and the *Precision Synthesis Style Guide* in the Precision RTL Synthesis Manuals Bookcase in the Help menu.

Incremental Compilation and Block-Based Design

As designs become more complex and designers work in teams, a block-based hierarchical or incremental design flow is often an effective design approach. In an incremental compilation flow, you can make changes to a part of the design while maintaining the placement and performance of unchanged parts of the design. Design iterations can be made dramatically faster by focusing new compilations on particular design partitions and merging results with the results of previous compilations of other partitions. In a bottom-up or team-based approach, you can perform optimization on individual blocks and then integrate them into a final design and optimize it at the top level.

Using the Precision RTL Synthesis software, you can create different netlist files for different partitions of a design hierarchy. This makes each partition independent of the others for either a top-down or a bottom-up incremental compilation flow. In either case, only the portions of a design

that have been updated must be recompiled during design iterations. You can make changes and resynthesize one partition in a design to create a new netlist without affecting the synthesis results or fitting of other partitions. The following steps show the general top-down compilation flow when using these features of the Quartus II software:

1. Create Verilog HDL or VHDL design files as you do in the regular design flow.
2. Determine which hierarchical blocks you want to treat as separate partitions in your design.
3. Create a project with multiple implementations (or create multiple projects) in the Precision RTL Synthesis software, one for each partition in the design.
4. Disable I/O pad insertion in the implementations for lower-level partitions.
5. Compile and synthesize each implementation or each project in the Precision RTL Synthesis software, and make constraints as in the regular design flow.
6. Import the EDIF netlist and the Tcl file for each partition into the Quartus II software and set up the Quartus II project(s) to use incremental compilation.
7. Compile your design in the Quartus II software and preserve the compilation results using the post-fit netlist type.
8. When you make design or synthesis optimization changes to part of your design, resynthesize only the changed partition to generate the new EDIF netlist and Tcl file. Do not resynthesize the implementations or projects for the unchanged partitions.
9. Import the new EDIF netlist and Tcl file into the Quartus II software and recompile the design in the Quartus II software using incremental compilation.



For more information about creating partitions and using the incremental compilation in the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Hierarchy and Design Considerations

To ensure the proper functioning of the synthesis flow, you can create separate partitions only for modules, entities, or existing netlist files. In addition, each module or entity must have its own design file. If two different modules are in the same design file but are defined as being part of different partitions, you cannot maintain incremental synthesis because both regions must be recompiled when you change one of the modules.

Altera recommends that you register all inputs and outputs of each partition. This makes logic synchronous and avoids any delay penalty on signals that cross partition boundaries.

If you use boundary tri-states in a lower level block, the Precision RTL Synthesis software pushes the tri-states through the hierarchy to the top level to make use of the tri-state drivers on output pins of Altera devices. Because pushing tri-states requires optimizing through hierarchies, lower level tri-states are not supported with a block-based compilation methodology. You should use tri-state drivers only at the external output pins of the device and in the top-level block in the hierarchy.



For more tips on design partitioning, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Creating a Design with Separate Netlist Files

The first step in a hierarchical or incremental design flow is to ensure that different parts of your design do not affect each other. Ensure that you have separate netlists for each partition in your design so that you can take advantage of the incremental compilation design flow in the Quartus II software. If the whole design is in one netlist file, changes in one partition affect other partitions because of possible node name changes when you resynthesize the design.

You can create different implementations for each partition in your Precision RTL project, which allows you to switch between partitions without leaving the current project file, or you can create a separate project for each partition if you need separate projects for a bottom-up or team-based design flow.

Create a separate implementation or a separate project for each lower level module and for the top-level design that you want to maintain as a separate EDIF netlist file. Implement black-box instantiations of lower level modules in your top-level implementation or project.

For more information about managing implementations and projects, refer to the *Precision RTL Synthesis User's Manual* in the Precision Manuals Bookcase in the Help menu.

When synthesizing the implementations for lower level modules, perform these steps:

1. On the **Tools** menu, turn off **Add IO Pads** on the **Optimization** page under **Set Options**.
2. Read the HDL files for the modules.

   Modules may include black-box instantiations of lower level modules that are also maintained as separate EDIF files.

3. Add constraints for all partitions in the design.

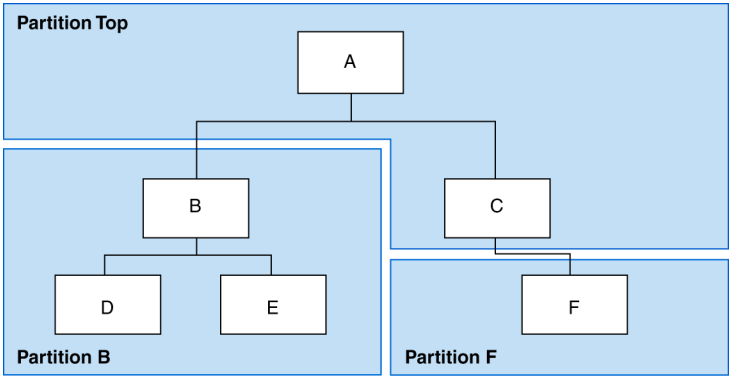
When synthesizing the top-level design implementation, perform these steps:

1. Read the HDL files for top-level designs.
2. Create black boxes for lower level modules in the top-level design.
3. Add constraints.

   In a top-down incremental compilation flow, constraints made on lower level modules are not passed to the Quartus II software. Ensure that appropriate constraints are made in the top-level Precision RTL Synthesis project, or in the Quartus II project.

The following sections describe an example of implementing black boxes to create separate EDIF netlists. Figure 10–2 shows an example of a design hierarchy separated into various partitions.

Figure 10–2. Partitions in a Hierarchical Design



In Figure 10–2, the top-level partition contains the top-level block in the design (block A) and the logic that is not defined as part of another partition. In this example, the partition for top-level block A also includes the logic in the C subblock. Because block F is contained in its own partition, it is not treated as part of the top-level partition A. Another separate partition, B, contains the logic in blocks B, D, and E. In a team-based design, different engineers may work on the logic in different partitions. One netlist is created for the top-level module A and its submodule C, another netlist is created for B and its submodules D and E, while a third netlist is created for F. To create multiple EDIF netlist files for this design, follow these steps:

1. Generate an EDIF file for module B. Use **B.v/.vhd**, **D.v/.vhd**, and **E.v/.vhd** as the source files.

2. Generate an EDIF file for module F. Use **F.v/.vhd** as the source file.

3. Generate a top-level EDIF file for module A. Use **A.v/.vhd** and **C.v/.vhd** as the source files. Ensure that you create black boxes for modules B and F, which were optimized separately in the previous steps.

Creating Black Boxes in Verilog HDL

Any design block that is not defined in the project or included in the list of files to be read for a project is treated as a black box by the software. In Verilog HDL, you must provide an empty module declaration for any module that is treated as a black box.

A black-box example for top-level file **A.v** follows. Use this same procedure for any lower level files, which also contain a black box for any module beneath the current level of hierarchy.

Example 10–20. Verilog HDL Black Box for Top-Level File A.v

```
module A (data_in, clk, e, ld, data_out);
    input data_in, clk, e, ld;
    output [15:0] data_out;

    wire [15:0] cnt_out;

    B U1 (.data_in (data_in),.clk(clk), .ld (ld),.data_out(cnt_out));
    F U2 (.d(cnt_out), .clk(clk), .e(e), .q(data_out));

// Any other code in A.v goes here.
endmodule

// Empty Module Declarations of Sub-Blocks B and F follow here.
// These module declarations (including ports) are required for black
// boxes.

module B (data_in, clk, ld, data_out);
    input data_in, clk, ld;
    output [15:0] data_out;
endmodule

module F (d, clk, e, q);
    input [15:0] d;
    input clk, e;
    output [15:0] q;
endmodule
```

Creating Black Boxes in VHDL

Any design block that is not defined in the project or included in the list of files to be read for a project is treated as a black box by the software. In VHDL, you need a component declaration for the black box just like any other block in the design.

A black box for the top-level file **A.vhd** is shown in the following example. Follow this same procedure for any lower level files that also contain a black box or for any block beneath the current level of hierarchy.

Example 10–21. VHDL Black Box for Top-Level File A.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY A IS
PORT ( data_in : IN INTEGER RANGE 0 TO 15;
    clk, e, ld : IN STD_LOGIC;
    data_out : OUT INTEGER RANGE 0 TO 15);
END A;

ARCHITECTURE a_arch OF A IS
COMPONENT B PORT(
    data_in : IN INTEGER RANGE 0 TO 15;
    clk, ld : IN STD_LOGIC;
    d_out : OUT INTEGER RANGE 0 TO 15);
END COMPONENT;

COMPONENT F PORT(
    d : IN INTEGER RANGE 0 TO 15;
    clk, e: IN STD_LOGIC;
    q : OUT INTEGER RANGE 0 TO 15);
END COMPONENT;

-- Other component declarations in A.vhd go here

signal cnt_out : INTEGER RANGE 0 TO 15;

BEGIN
U1 : B
PORT MAP (
    data_in => data_in,
    clk => clk,
    ld => ld,
    d_out => cnt_out);

U2 : F
PORT MAP (
    d => cnt_out,
    clk => clk,
    e => e,
    q => data_out);

-- Any other code in A.vhd goes here

END a_arch;
```

After you complete the steps outlined in this section, you have different EDIF netlist files for each partition of the design. These files are ready for use in the incremental compilation or LogicLock design methodologies in the Quartus II software.

Creating Quartus II Projects for Multiple EDIF Files

The Precision RTL Synthesis software creates a Tcl file for each EDIF file, and provides the Quartus II software with the appropriate constraints and information to set up a project. For details about using the Tcl script generated by the Precision RTL software to set up your Quartus II project and to pass your top-level constraints, refer to "Running the Quartus II Software Manually Using the Precision RTL Synthesis-Generated Tcl Script" on page 10–16.

Depending on your design methodology, you can create one Quartus II project for all EDIF netlists (a top-down flow), or a separate Quartus II project for each EDIF netlist (a bottom-up flow). In a top-down compilation design flow, you create design partition assignments and floorplan location assignments for each partition in the design within a single Quartus II project. This methodology provides the best quality of results and performance preservation during incremental changes to your design. You may need to use a bottom-up design flow when each partition must be optimized separately, such as in certain team-based design flows.

To perform a bottom-up compilation in the Quartus II software, create separate Quartus II projects and import each design partition into a top-level design using the incremental compilation export and import features to maintain placement results. Alternately, you can use the LogicLock design methodology to import each lower-level partition and maintain placement results.

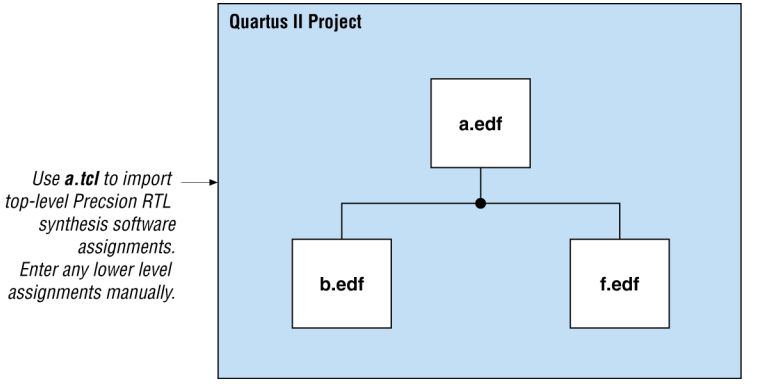
The following sections describe how to create the Quartus II projects for these two design flows.

Creating a Single Quartus II Project for a Top-Down Incremental Compilation Flow

Use the <*top-level project*>**.tcl** file generated for the top-level partition to create your Quartus II project and import all the netlists into this one Quartus II project for an incremental compilation flow. You can optimize all partitions within the single Quartus II project and take advantage of the performance preservation and compilation time reduction that incremental compilation provides. Figure 10–3 shows the design flow for the example design in Figure 10–2 on page 10–36.

All the constraints from the top-level implementation are passed to the Quartus II software in the top-level Tcl file, but any constraints made only in the lower level implementations within the Precision RTL Synthesis software are not forward-annotated. Enter these constraints manually in your Quartus II project.

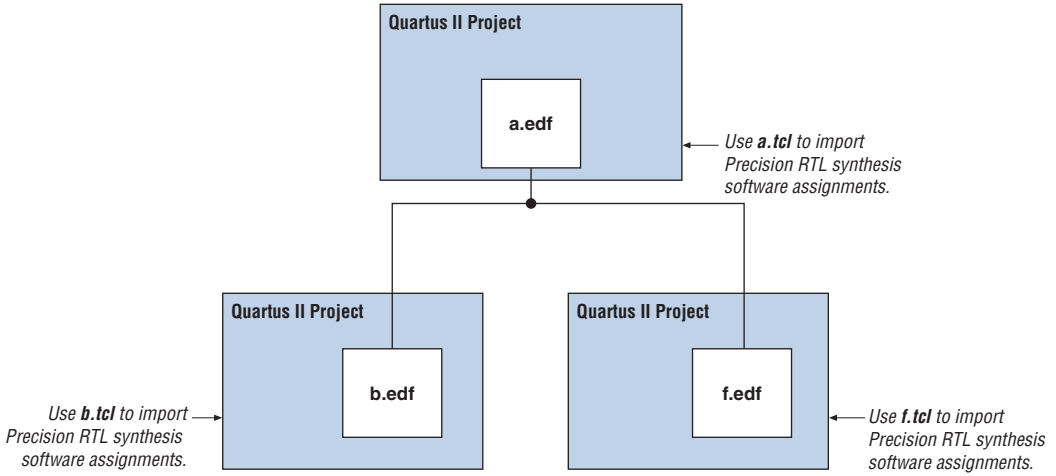
Figure 10–3. Design Flow Using Multiple EDIF Files with One Quartus II Project



Creating Multiple Quartus II Projects for a Bottom-Up Flow

Use the Tcl files generated by the Precision RTL Synthesis software for each Precision RTL Synthesis software implementation or project to generate multiple Quartus II projects, one for each partition in the design. Each designer in the project can optimize their block separately in the Quartus II software and export the placement of their blocks using the incremental compilation or LogicLock design methodology. Designers should create a LogicLock region for each block; the top-level designer should then import all the blocks and assignments into the top-level project. Figures 10–4 shows the design flow for the example design in Figure 10–2 on page 10–36.

Figure 10–4. Design Flow: Using Multiple EDIF Files with Multiple Quartus II Projects



Conclusion

Advanced synthesis is an important part of the design flow. The Mentor Graphics Precision RTL Synthesis software and Quartus II design flow allows you to control how to prepare your design files for the Quartus II place-and-route process. This allows you to improve performance and optimize a design for use with Altera devices. Several of the methodologies outlined in this chapter can help you optimize a design to achieve performance goals and decrease design time.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *Precision RTL Synthesis User's Manual* in the Precision Manuals Bookcase in the Help menu
- *Precision Synthesis Style Guide* in the Precision RTL Synthesis Manuals Bookcase in the Help menu
- *Precision Synthesis Reference Manual* in the Precision Manuals Bookcase in the Help menu
- *Specifying EDA Tool Settings* in the Quartus II Help index
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*

Document Revision History

Table 10–7 shows the revision history for this chapter.

Table 10–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October, 2007 v7.2.0	• Added Arria GX to the list of supported devices • Added set_max_delay constraint • Added set_min_delay constraint	Updated document based on the Quartus II software version 7.2
May 2007 v7.1.0	• Minor updates for the Quartus II software version 7.1. • Add "Referenced Documents" section	—
March 2007 v7.0.0	• Chapter 10 was formally Chapter 9 in version 6.0. • Added SDC support for Stratix III and Cyclone III devices	Added information regarding SDC for Stratix III and Cyclone III; updated information about in Precision RTL Synthesis software and its compatibility with the Quartus II software.
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.	—

Table 10–7. Document Revision History (Continued)

Date and Document Version	Changes Made	Summary of Changes
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 9 was formerly Chapter 10 in version 5.0.	—
May 2005 v5.0.0	Chapter 10 was formerly chapter 8 in version 4.2.	—
December 2004 v2.1	• Chapter 9 was formerly Chapter 10 in version 4.1. • Updates to tables and figures. • New functionality for Quartus II software version 4.2.	—
June 2004 v2.0	• Updates to tables and figures. • New functionality for Quartus II software version 4.1.	—
February 2004 v1.0	Initial release.	—

11. Synopsys Design Compiler FPGA Support

QII51014-7.2.0

Introduction

Programmable logic device (PLD) designs have reached the complexity and performance requirements of ASIC designs. As a result, advanced synthesis has taken on a more important role in the design process. This chapter documents the usage and design flow of the Synopsys Design Compiler FPGA (DC FPGA) synthesis software with Altera® devices and Quartus® II software. DC FPGA supports Stratix® II, Stratix, Stratix GX, Cyclone® II, and Cyclone devices.

This chapter assumes that you have set up and licensed the DC FPGA software and Altera Quartus II software.

This chapter is primarily intended for ASIC designers experienced with the Design Compiler (DC) software who are now developing PLD designs, and experienced PLD designers who would like an introduction to the Synopsys DC FPGA software.



To obtain the DC FPGA software, libraries, and instructions on general product usage, go to the Synopsys website at **http://solvnet.synopsys.com/retrieve/012889.html**

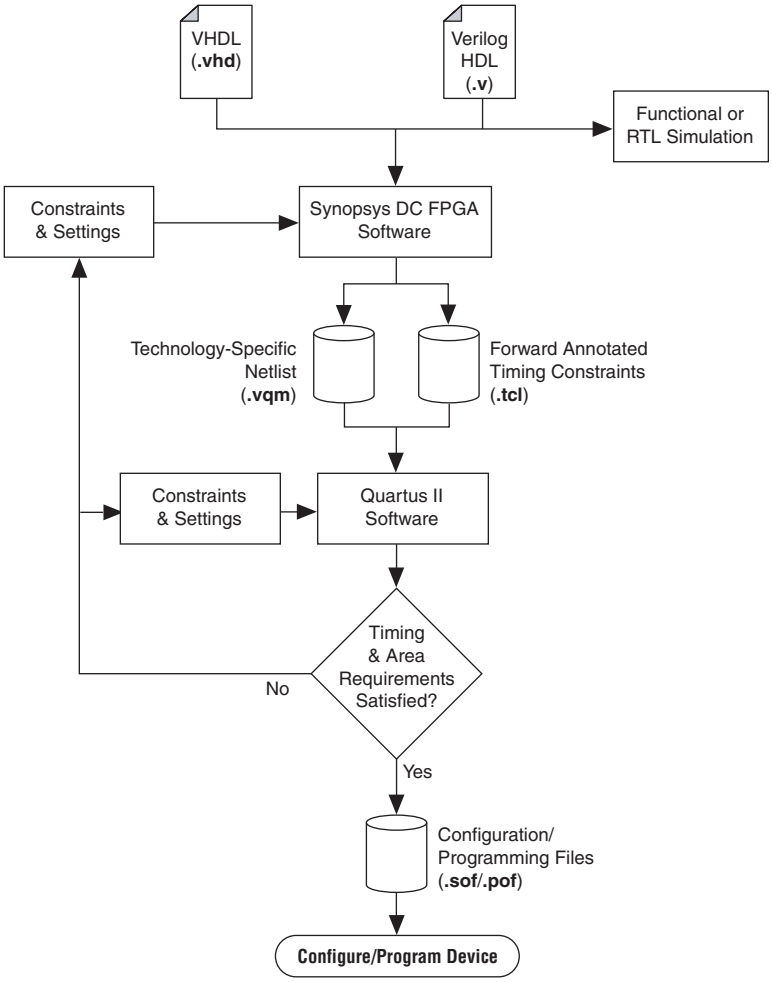
The following areas are covered in this chapter:

- General design flow with the DC FPGA software and the Quartus II software
- Initialization procedure using the **.synopsys_dc.setup** file for targeting Altera devices
- Using Altera megafunctions with the DC FPGA software
- Reading design files into the DC FPGA software
- Applying synthesis and timing constraints
- Reporting and saving design information
- Exporting designs to the Quartus II software

Design Flow Using the DC FPGA Software and the Quartus II Software

A high-level overview of the recommended design flow for using the DC FPGA software with the Quartus II software is shown in Figure 11–1.

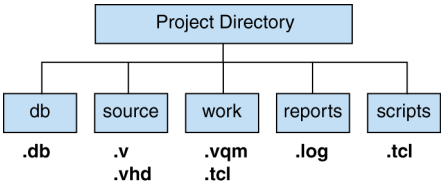
Figure 11–1. Design Flow Using the DC FPGA Software and the Quartus II Software



Setup of the DC FPGA Software Environment for Altera Device Families

Altera recommends that you organize your project directory with several subdirectories. A recommended project hierarchy is shown in Figure 11–2.

Figure 11–2. Project Hierarchy



To use the DC FPGA software to synthesize HDL designs for use with the Quartus II software, the required settings should be included in your **.synopsys_dc.setup** initialization file. This file is used to define global variables and direct the DC FPGA software to the proper libraries used for synthesis, as well as set internal assignments for synthesizing designs for Altera devices.

The **.synopsys_dc.setup** file can reside in any one of three locations and be read by the DC FPGA software. The DC FPGA software automatically reads the **.synopsys_dc.setup** file at startup in the following order of precedence:

1. Current directory where you run the DC FPGA software shell.
2. Home directory.
3. The DC FPGA software installation directory.

The DC FPGA software has vendor-specific setup files for each of the Altera logic families in the installation directory. These vendor-specific setup files are found where you have installed the libraries (*<dcfpga_rootdir>***/libraries/fpga/altera**) and are named in the form **synopsys_dc_***<logic family>***.setup**. For example, if you want to use the default setup for synthesizing an Altera Stratix device, you must link to or copy the **synopsys_dc_stratix.setup** to your home or current directory and rename the file **.synopsys_dc.setup**.

Synopsys recommends using the vendor-specific setup files provided with each release of the DC FPGA software to ensure that you have all the correct settings and obtain the best quality results.

Example 11–1 contains the recommended synthesis settings for the Stratix II device architecture.

Example 11–1. Recommended Synthesis Settings for Stratix II Device Architecture

```
# Setup file for Altera Stratixii
# TCL style setup file but will work for original DC shell as well
# Need to define the root location of the libraries by chaning the variable
$dcfpga_lib_path

set dcfpga_lib_path "<dcfpga_rootdir>/libraries/fpga/altera"


set search_path ". $dcfpga_lib_path $dcfpga_lib_path/STRATIXII $search_path"
set target_library "stratixii.db"
set synthetic_library "tmg.sldb altera_mf.sldb lpm.sldb"
set link_library "* stratixii.db tmg.sldb altera_mf.sldb lpm.sldb stratixii_mf.sldb"

set_fpga_defaults altera_stratixii
```

After generating your **.synopsys_dc.setup** file, run the DC FPGA software in either the Tcl shell or in the Design Compiler software shell without Tcl support. Run the DC FPGA software shell at a command prompt by typing `fpga_shell-t` or `fpga_shell -tcl` for the Tcl shell version of the DC FPGA software. Run the non-Tcl version of the DC FPGA software with the `fpga_shell` command. Altera recommends using the Tcl shell for all of your synthesis work.

If you have created a Tcl synthesis script for use in the DC FPGA software and wish to run it immediately at startup, you can start the DC FPGA software shell and run the script with the command shown in the example below:

`fpga_shell-t -f` *<path>/<script filename>*`.tcl` ↵

Otherwise, you can run your scripts at any time at the `fpga_shell-t>` prompt with the `source` command. An example is shown below:

`source` *<path>/<script filename>*`.tcl` ↵

Megafunctions and Architecture-Specific Features

Altera provides parameterized megafunctions including library of parameterized modules (LPMs), device-specific Altera megafunctions, intellectual property (IP) available as Altera MegaCore® functions, and IP available through the Altera Megafunction Partners Program (AMPP). You can use megafunctions by instantiating them in your HDL code, or by inferring them from your HDL code during synthesis in the DC FPGA software.



For more details on specific Altera megafunctions, refer to the Quartus II Help.

The DC FPGA software automatically recognizes certain types of HDL code and infers the appropriate megafunction when a megafunction provides optimal results. The DC FPGA software also provides options to control inference of certain types of megafunctions, as described in the section "Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager" on page 11–6.



For a detailed discussion about instantiating versus inferring megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. This chapter also provides details about using the MegaWizard® Plug-In Manager in the Quartus II software and explains the files generated by the wizard. In addition, the chapter provides coding style recommendations and examples for inferring megafunctions in Altera devices.

If you instantiate a megafunction in your HDL code, you can use the MegaWizard Plug-In Manager to parameterize the function, or you can instantiate the function using the port and parameter definition. The MegaWizard Plug-In Manager provides a graphical interface in the Quartus II software for customizing and parameterizing megafunctions. "Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager" on page 11–6 describes the MegaWizard Plug-In Manager flow with the DC FPGA synthesis software.

Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager

When you use the MegaWizard Plug-In Manager to set up and parameterize a megafunction, the MegaWizard Plug-In Manager creates a VHDL or Verilog HDL wrapper file that instantiates the megafunction (a black box methodology). The MegaWizard can also generate a fully elaborated netlist that is read by EDA synthesis tools, such as the DC FPGA (a clear box methodology). Both clear box and black box methodologies are described in the following sections.

Clear Box Methodology

You can use the MegaWizard Plug-In Manager to generate a fully synthesizeable netlist. This flow is referred to as a clear box methodology because starting in V-2005.06, the DC FPGA software can look into the megafunction file. The clear box feature enables the synthesis tool to report more accurate timing estimates and resource utilization, while taking a better advantage of timing driven optimization than a black box methodology.

This clear box feature is enabled by turning on the **Generate clear box netlist file instead of a default wrapper file (for use with supported EDA synthesis tools only)** option in the MegaWizard Plug-In Manager for certain megafunctions. DC FPGA supports clear box megafunctions for `altmult_add`, `almult_accum`, `altsyncram` and `altshift_taps`. If the option does not appear, then clear box models are not supported for the selected megafunction.



The library declarations in the MegaWizard generated VHDL output files need to be manually commented out to work properly with the DC FPGA.

Reading Megafunction Wizard-Generated Synthesizable Clear Box Netlist Files for Megafunction Instantiation

The DC FPGA software analyzes and elaborates the Megafunction Wizard-generated Verilog HDL <*output file*>**.v** or VHDL <*output file*>**.vhd** netlist that contains the parameters needed by the Quartus II software to properly configure and instantiate your megafunction. Analyze the clear box netlist files along with the rest of the RTL files during synthesis in DC FPGA. The resulting netlist contains all the primitives that are part of the clear box netlist. There is no need to put the clear box netlist file in your Quartus II project along with your DC FPGA generated netlist file.

Using the clear box Megafunction Wizard-generated netlist files provides the DC FPGA software an understanding of their timing arcs and resource usage. The DC FPGA software uses timing information to optimize the surrounding circuits and resource data to better manage the overall resource usage for the whole design. The DC FPGA software takes the clear box netlist timing and area data into account when reporting the timing and resource utilization for the device.

Advanced Clear Box Support for the Direct-Instantiated or Inferred Clear Box Megafunctions

The DC FPGA provides advanced clear box support that enables a clear box implementation for the direct-instantiated or inferred megafunctions in your design. This methodology allows the DC FPGA to obtain the most accurate interface timing and area data for the megafunctions. Therefore, synthesis optimization is more effective, and timing and area reports are more accurate.

The following describes the setup and usage model for this advanced clear box support.

Design Compiler FPGA Setup

The advanced clear box flow will be enabled in the DC FPGA only when the **clearbox.sldb** synthetic library is added to the `synthetic_library` variable. For example:

```
set synthetic_library [concat clearbox.sldb $synthetic_library]
set link_library [concat clearbox.sldb $link_library]
```

Specify the path to the clear box loader (executable) in one of the following ways:

- Set the `synlib_cbx_exec_path` variable to the absolute path of the clear box loader before the compile command:
  `set synlib_cbx_exec_path` <*Quartus II installation directory /bin/clearbox*>
- Set the UNIX environment variable `CLEARBOX_EXEC_PATH` to the absolute path of the clear box loader. For example:
  `setenv CLEARBOX_EXEC_PATH` <*Quartus II installation directory /bin/clearbox*>

By default, the advance clear box flow is turned off. To enable the clear box advanced flow, add the following to your DC FPGA script. Set it before the compile command:

```
set fpga_altera_clearbox_for_user_cells true
```

UNIX Environment Setting

For the DC FPGA to work with the clear box loader, the following setting is necessary for the LD_LIBRARY_PATH environment variable. Assume the QuartusII_Path used below is set to the Quartus II installation directory.

On a Linux platform:

```
setenv LD_LIBRARY_PATH QuartusII_Path/linux:$LD_LIBRARY_PATH
```

On a Solaris platform:

```
setenv LD_LIBRARY_PATH QuartusII_Path/solaris:$LD_LIBRARY_PATH
```

Error Message

The only error message that you might encounter when trying to enable the advanced clear box flow is: DCFPGA_UEGI-1

The DC FPGA reports this error when one of the following situations occurs:

- It cannot find the clear box loader path. For example, the defined path is incorrect.
- The Loader is not found in the specified path.
- The Loader specified is not executable.

Sample Design Compiler FPGA Clear Box Setup Script

The TCL script shown in Example 11–2 is a DC FPGA clear box setup script. Use it before compiling the design in DC FPGA.

Example 11–2. Sample Clear Box Setup Script

```
set QuartusII_Path /tools/altera/qii51
set_unix_variable CLEARBOX_EXEC_PATH $QuartusII_Path/bin/clearbox
set old_llp [get_unix_variable LD_LIBRARY_PATH]
set platform [sh uname]

if { $platform == "Linux" } {
    set_unix_variable LD_LIBRARY_PATH $QuartusII_Path/linux: old_llp
} else {
    # Assume, if not linux, it is solaris
    set_unix_variable LD_LIBRARY_PATH $QuartusII_Path/solaris: old_llp

set synthetic_library [concat clearbox.sldb $synthetic_library]
set link_library [concat clearbox.sldb $link_library]

set fpga_altera_clearbox_for_user_cells true
```

Black Box Methodology

Using the MegaWizard Plug-In Manager-generated wrapper file is referred to as a black box methodology because the megafunction is treated as a black box in the DCFPGA software. The black box wrapper file is generated by default in the MegaWizard Plug-In Manager and is available for all megafunctions. The black box methodology does not allow the synthesis tool any visibility into the function module and therefore, does not take full advantage of the synthesis tool's timing driven optimization.

There are two ways of instantiating Megafunction Wizard-generated functions in your design hierarchy loaded in the DC FPGA software. You can instantiate and compile the Verilog HDL or VHDL variation wrapper file description of your megafunction in the DC FPGA software, or you can instantiate a black box that just describes the ports of your megafunction variation wrapper file.



The library declarations in the MegaWizard generated VHDL output files need to be manually commented out to work properly with the DC FPGA.

Reading Megafunction Wizard-generated Variation Wrapper Files

The DC FPGA software has the ability to analyze and elaborate the Megafunction Wizard-generated Verilog HDL <*output file*>**.v** or VHDL <*output file*>**.vhd** netlist that contains the parameters needed by the Quartus II software to properly configure and instantiate your megafunction. The DC FPGA software may take advantage of this variation wrapper file during the optimization of your design to reduce area utilization and improve path delays. DC FPGA also supports altpll in a non-black box flow (that is, the DC FPGA can automatically derive PLL output clocks when the user has specified only the PLL input clock).

Using the megafunction variation wrapper file <*output file*>**.v** or <*output file*>**.vhd** in the DC FPGA software synthesis provides good synthesis results for area estimates, but actual timing results are best predicted after place-and-route inside the Quartus II software. However, reading the megafunction variation wrapper allows the DC FPGA software to provide better synthesis estimates over a black box methodology.

Using Megafunction Wizard-Generated Variation Wrapper Files in a Black Box Methodology

Instantiating the megafunction wizard-generated wrapper file without reading it in the DC FPGA software is referred to as a black box methodology because the megafunction is treated as an unknown container in the DC FPGA software.

The black box methodology does not allow synthesis software to have any visibility into the module, thereby not taking full advantage of the timing driven optimization of the DC FPGA software and preventing the software from estimating logic resources for the black box design.

Using Megafunction Wizard-Generated Verilog HDL Files for Black Box Megafunction Instantiation

By default, the MegaWizard Plug-In Manager generates the Verilog HDL instantiation template file <*output file*>**_inst.v** and the black box module declaration <*output_file*>**_bb.v** for use in your design in the DC FPGA software. The instantiation template file helps to instantiate the megafunction variation wrapper file, <*output file*>**.v**, in your top-level design. Do not include the megafunction variation wrapper file in the DC FPGA software project if you are following the black box methodology. Instead, add the wrapper file and your generated Verilog Quartus Mapping (**.vqm**) netlist in your Quartus II project. Add the hollow body black box module declaration <*output file*>**_bb.v** to your linked design files in the DC FPGA software to describe the port connections of the black box.

Using Megafunction Wizard-Generated VHDL Files for Black Box Megafunction Instantiation

By default, the MegaWizard Plug-In Manager generates a VHDL component declaration file <*output file*>**.cmp** and a VHDL instantiation template file <*output file*>**_inst.vhd** for use in your design. These files can help you instantiate the megafunction variation wrapper file, <*output file*>**.vhd**, in your top-level design. Do not include the megafunction variation wrapper file in the DC FPGA software project. Instead, add the wrapper file and your generated Verilog Quartus Mapping netlist in your Quartus II project.



The DC FPGA software supports direct instantiation of all LPMs and megafunctions. For a complete list of all LPMs and Megafunctions, refer to the following two files in your Quartus II installation directory:

- <Quartus II installation directory> **/libraries/vhdl/lpm/lpm_pack.vhd**
- <Quartus II installation directory> **/libraries/vhdl/altera_mf/altera_mf_components.vhd**

DC FPGA supports direct instantiation of LPMs and megafunctions only. These macro functions include all Altera IP cores and all components listed in:

<***Quartus II installation directory***>**/libraries/vhdl/altera_mf_components.vhd** or **stratixgx_mf_components.vhd**.

The following example is the usage model using the **mypll** for direct instantiation:

1. During synthesis in DC FPGA, analyze the variation file **mypll.[v|vhd]** along with the rest of the RTL files.

2. During place-and-route in the Quartus II software, simply run the self-contained Verilog Quartus Mapping File. You do not need to put the variation file in the Verilog Quartus Mapping directory.

The benefit of using the direct instantiation method is that the DC FPGA is able to utilize the available clock enable pins of the LPMs and megafunctions during the automatic gated-clock conversion process.

Inferring Altera Megafunctions from HDL Code

The DC FPGA software automatically recognizes certain types of HDL code, and maps digital signal processing (DSP) functions and memory (RAM and ROM) to efficient, technology-specific implementations. This allows the use of technology-specific resources to implement these structures by inferring the appropriate Altera megafunction when it provides optimal results.



For coding style recommendations and examples for inferring megafunctions in Altera devices, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Depending on the coding style, if you do not adhere to these recommended HDL coding style guidelines, it is possible that the DC FPGA software and Quartus II software will not take advantage of the high performance DSP blocks and RAMs, and may instead

implement your logic using regular logic elements (LEs). This causes your logic to consume more area in your device and may adversely affect your design performance. Altera device families do not all share the same resources, so your HDL coding style may cause your logic to be implemented differently in each family. For example, Stratix devices contain dedicated DSP blocks which Cyclone devices lack. In a Cyclone device, multipliers are implemented in LEs.

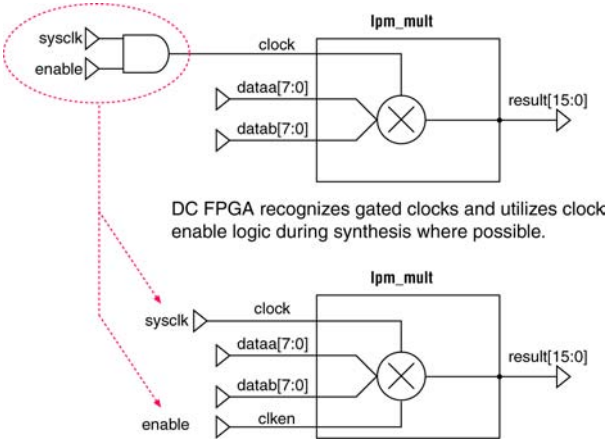
Example 11–3 shows Verilog HDL code that infers a two-port RAM that can be synthesized into an M512 RAM block of a Stratix device.

Example 11–3. Verilog HDL Code Inferring a Two-Port RAM

```
module example_ram (clk, we, rd_addr, wr_addr, data_in, data_out);
input clk, we;
input [15:0] data_in;
output [15:0] data_out;
input [7:0] rd_addr;
input [7:0] wr_addr;
reg [15:0] ram_data [7:0];
reg [15:0] data_out_reg;
always @ (posedge clk)
begin
if (we)
        ram_data[wr_addr] <= data_in;
data_out_reg <= ram_data[rd_addr];
end
assign data_out = data_out_reg;
endmodule
```

One of the strengths of the DC FPGA software is its gated clock conversion feature. Inferring megafunctions in HDL takes advantage of this feature. For gated clocks or clock enables designed outside of LPMs, Altera-specific megafunctions, and registers, the DC FPGA software merges the gated clock functions into these design elements using dedicated clock enable functionality during synthesis. The DC FPGA software reconfigures the megafunction block or register to synthesize the clock enable control logic. This can save area in your design and improve your design performance by reducing the gated clock path delay and the amount of logic used to implement the design. An illustration of this kind of gated clock optimization is shown in Figure 11–3.

Figure 11–3. Gated Clock Optimization



The DC FPGA software does not perform gated clock optimization on instantiated black box megafunctions or on instantiated megafunction variation wrapper file. The DC FPGA software performs gated clock optimization only on synthesizable inferred megafunctions.

Reading Design Files into the DC FPGA Software

The process of reading design files into the DC FPGA software is a two-step process where the DC FPGA software analyzes your HDL design for syntax errors, then elaborates the specified design. The elaboration process finds analyzed designs and instantiates them in the elaborated design's hierarchy. You must identify which supported language the files are written in when reading designs into the DC FPGA software. The supported HDL languages are listed in Table 11–1.

Table 11–1. Supported Design File Formats

Format	Description	Keyword	Extension
Verilog HDL (Synopsys Presto HDL)	Verilog hardware description language	verilog	.v
VHDL	VHSIC hardware description language	vhdl	.vhd
.db	Synopsys internal database format *(1)*	db	.db
EDIF	Electronic design interchange format	edif	.edf

Note to Table 11–1:
(1) The Design Compiler DB format file requires additional license keys.

To set most of the required synthesis settings to generate an optimal netlist, use the following command:

```
set_fpga_defaults <architecture_name>
```

For example:

```
set_fpga_defaults altera_stratixii
```

Use the following commands to analyze and elaborate HDL designs in the DC FPGA software:

```
analyze -f <verilog|vhdl> <design file> ↵
```

```
elaborate <design name> ↵
```

Once a design is analyzed, it is stored in a Synopsys library format file in your working directory for reuse. You need to re-analyze the design only when you change the source HDL file. Elaboration is performed after you have analyzed all of the subdesigns below your current design.

Another way to read your design is by using the `read_file` command. This can be used to read in gate-level netlists that are already mapped to a specific technology. The `read_file` command performs analysis and elaboration on Verilog HDL and VHDL designs that are written in register transfer level (RTL) format. The difference between the `read_file` command and the analyze and elaborate combination is that the `read_file` command elaborates every design read, which is unnecessary. Only the top-level design must be elaborated. The `read_file` command is useful if you have a previously synthesized block of logic that you want to re-use in your design.

To use the `read_file` command for a specific language, type the following command:

```
read_file -f <verilog|vhdl|db|edif> <design file> ↵
```

You can also read files in specific languages using the `read_verilog`, `read_vhdl`, `read_db`, and `read_edif` commands.

Once you have read all of your design files, specify the design you want to focus your work on with the `current_design` command. This is usually the top module or entity in your design that you wish to compile up to. To use this command, type the following:

```
current_design <design name> ↵
```

You then need to build your design from all of the analyzed HDL files with the link command. To use this command, type the following:

```
link ↵
```

After linking your designs successfully in the DC FPGA software, you should specify the constraints you are applying to your design. In the DC FPGA software, you have the capability of loading multiple levels of hierarchy and synthesizing specific blocks in a bottom-up synthesis methodology, or you can synthesize the entire design from the top-level module in a top-down synthesis methodology.

You can switch the current focus of the DC FPGA software between the designs loaded by using the `current_design` command. This changes your current focus onto the design specified, and all subsequent constraints and commands will apply to that design.

If you have read Quartus II megafunction wizard-generated designs or third-party IP into the DC FPGA software, you can instruct the DC FPGA software not to synthesize the IP. Use the `set_dont_touch` constraint and apply it to each module of your design that you do not want synthesized. To use this command, type the following:

`set_dont_touch` <*design name*> ↵

Using the `set_dont_touch` command can be helpful in a bottom-up synthesis methodology, where you optimize designs at the lower levels of your hierarchy first and do not allow the DC FPGA software to resynthesize them later during the top-level integration. However, depending on the design's HDL coding, you might want to allow top-level resynthesis to get further area reduction and improved path delays. For best results, Altera recommends following the top-down synthesis methodology and not using the `set_dont_touch` command on lower level designs.

Selecting a Target Device

If you do not select an Altera device, the DC FPGA software, by default, synthesizes for the fastest speed grade of the logic family library that is loaded in your **.synopsys_dc.setup** file. If you are targeting a specific device of an Altera family, you must have the correct library linked, then specify the device for synthesis with the `set_fpga_target_device` command**.** To use this command, type the following:

`set_fpga_target_device` <*device name*> ↵

You can have the DC FPGA software produce a list of all available devices in the linked library by adding the `-show_all` option to the `set_fpga_target_device` command. An example of this list of devices for the Stratix II library is shown in Example 11–4.

Example 11–4. List of Available Devices in the Linked Library Using the -show_all Option

```
Loading db file '/dc_fpga/libraries/fpga/altera/STRATIXII/stratixii.db'

  Valid device names are:

Part              Pins      FFs     Speed Grades
-------------------------------------------------
AUTO *            0         0         FASTEST
EP2S15F484        484       12480     C4
EP2S15F672        672       12480     C4
EP2S30F484        484       27104     C4
EP2S30F672        672       27104     C4
EP2S60F484        484       48352     C4
EP2S60F672        672       48352     C4
EP2S60F1020       1020      48352     C4
EP2S90F1020       1020      72768     C4
EP2S90F1508       1508      72768     C4
EP2S130F1020      1020      106032    C4
EP2S130F1508      1508      106032    C4
EP2S180F1020      1020      143520    C4
EP2S180F1508      1508      143520    C4

  * Default part
```

For example, if you want to target the C4 speed grade device of the Stratix II EP2S60F672 device, apply the following constraint:

```
set_fpga_target_device EP2S60F672C4
```

Timing and Synthesis Constraints

You must create timing and synthesis constraints for your design for the DC FPGA software to optimize your design performance. The timing constraints specify your desired clocks and their characteristics, input and output delays, and timing exceptions such as false paths and multi-cycle paths. The synthesis constraints define the device, the type of I/O buffers that should be used for top-level ports, and the maximum register fan-out threshold before buffer insertion is performed. Synopsys Design Constraints (SDCs) are Tcl-format commands that are widely used in many EDA software applications. The DC FPGA software supports the same SDC commands that the full version of the Design Compiler software supports. However, certain constraints that are used in ASIC synthesis are not applicable to programmable logic synthesis, so the DC FPGA software ignores them.

The DC FPGA software supports the following constraints:

- `create_clock`
- `set_max_delay`
- `set_propagated_clock`
- `set_input_delay`
- `set_output_delay`
- `set_multicycle_path`
- `set_false_path`
- `set_disable_timing`
- `set_fpga_resource_limit`
- `set_register_max_fanout`
- `set_max_fanout`
- `set_fpga_target_device`



For the syntax and full usage of these commands, refer to the *Synopsys DC FPGA User Guide*.



For synthesis with the DC FPGA software, minimum timing analysis is not necessary, as it primarily looks at setup timing optimization to achieve the fastest clock frequency for your design. Altera recommends adding additional minimum timing constraints to your design inside the Quartus II software.

The DC FPGA forward annotates all the clock, timing exceptions, and I/O delay constraints to Quartus II when the `write_par_constraint` command is used in the DC FPGA. For more information about this command, refer to "Exporting Designs to the Quartus II Software" on page 11–22. Since the Quartus II software does not support the `through` option for the timing exception constraints, the DC FPGA does not forward annotate constraints that use the `through` option.

In the DC FPGA software, timing constraints applied to inferred RAM, ROM, shift registers, and DSP MAC functions are obeyed. However, these constraints are not forward-annotated to the Quartus II software because these functions are inferred to Altera megafunctions. The Quartus II software does not support timing constraints applied to megafunctions. The workaround is to run the Verilog Quartus Mapping/EDIF netlist through analysis and synthesis in the Quartus II software (**quartus_map**). All megafunctions expand to atom primitives. These atom primitives can be processed by the Quartus II software. You can then apply constraints to the internal atoms of the megafunctions.

The timing reports generated from the DC FPGA software are preliminary estimates of the path delays in your design, and accurate timing is reported only after place-and-route is performed with the Quartus II software.

The DC FPGA software also performs cross-hierarchical boundary optimization. Altera recommends running this command before a compilation:

```
ungroup -small 500 ↵
```

This allows the DC FPGA software to potentially improve area reduction and performance improvement by ungrouping smaller blocks of logic in your design hierarchy and combining functions.

Compilation and Synthesis

After applying timing and synthesis constraints, you can begin the compilation and synthesis process. The `compile` command runs this process within the DC FPGA software. To run a compilation, at the shell prompt type:

```
compile ↵
```

The compilation process performs two kinds of optimization:

- Architectural optimization focuses on the HDL description and performs high-level synthesis tasks such as sharing resources and sub-expressions, selecting Synopsys Design Ware implementations, and re-ordering operators.
- Gate-level optimization works on the generic netlist created by logic synthesis and works to improve the mapping efficiency to save area and improve performance by minimizing path delays.

Compilation can be done using a top-down synthesis methodology or a bottom-up synthesis methodology. The top-down synthesis methodology involves a single compilation of your entire design with the focus on the top module or entity of your design. The bottom-up synthesis methodology involves incremental compilation of major blocks in your design hierarchy and top-level integration and optimization. Either methodology can be applied when synthesizing for Altera devices. For best results, Altera recommends following the top-down synthesis methodology.

An example synthesis script that reads the design, applies timing constraints, reports results, saves the synthesized netlist file in the Verilog Quartus Mapping File format, and creates the Tcl scripts to work with the

Quartus II software is shown in Example 11–5. It uses the command `write_fpga`, which is described in "write_fpga Command" on page 11–22.

Example 11–5. Sample Synthesis Script

```
# Setup output directories
set outdir ./design
file delete -force $outdir
file mkdir $outdir
set rptdir ./report
file delete -force $rptdir
file mkdir $rptdir
#  Enable Presto compiler for VHDL design files
#  set hdlin_enable_presto_for_vhdl TRUE
# Setup libraries
define_design_lib work-path .$outdir/work
file mkdir $outdir/work
analyze -format verilog ./source/mult_box.v
analyze -format verilog ./source/mult_ram.v
analyze -format verilog ./source/top_module.v
elaborate top_module
link
current_design top_module
create_clock -period 5 [get_ports clk]
set_input_delay -max 2 -clock clk [get_ports {data_in_* mode_in}]
set_input_delay -min 0.5 -clock clk [get_ports {data_in_* mode_in}]
set_output_delay -max 2 -clock clk [get_ports {data_out ram_data_out_port} ]
set_output_delay -min 0.5 -clock clk [get_ports {data_out ram_data_out_port} ]
set_false_path -from [get_ports reset]
ungroup -small 500
compile
report_timing > $rptdir/top_module.log
report_fpga > $rptdir/top_module_fpga.log
write_fpga $outdir
quit
```

Reporting Design Information

After compilation is complete, the DC FPGA software reports information about your design. You can specify which kinds of reports you want generated with the reporting commands shown in Table 11–2.

Table 11–2. Reporting Commands

Object	Command	Description
Design	report_design	Reports design characteristics
	report_area	Reports design size and object counts
	report_hierarchy	Reports design hierarchy
	report_resources	Reports resource implementations
	report_fpga	Reports FPGA resource utilization statistics for the design
Instances	report_cell	Displays information about instances
References	report_reference	Displays information about references
Ports	report_port	Displays information about ports
	report_bus	Displays information about bused ports
Nets	report_net	Reports net characteristics
	report_bus	Reports bused net characteristics
Clocks	report_clock	Displays information about clocks
Timing	report_timing	Checks the timing of the design
	report_constraint	Checks the design constraints
	check_timing	Checks for unconstrained timing paths and clock-gating logic
	report_design	Shows operating conditions, timing ranges, internal input and output, and disabled timing arcs
	report_port	Shows unconstrained input and output ports and port loading
	report_timing_requirements	Shows all timing exceptions set on the design
	report_clock	Checks the clock definition and clock skew information
	derive_clocks	Checks internal clock and unused registers
	report_path_group	Shows all timing path groups in the design
Cell Attributes	get_cells	Shows all cell instances that have a specific attribute



For more information about the usage of these commands, refer to the *Synopsys DC FPGA User Guide*.

The DC FPGA software only provides preliminary estimates of your design's timing delays because the timing of your design cannot be accurately predicted until the Quartus II software has placed and routed your design.

Saving Synthesis Results

After synthesis, the technology-mapped design can be saved to a file in one of the following four formats: Verilog HDL, VHDL, Synopsys internal DB, or EDIF.

The Quartus II software accepts an EDIF netlist or Verilog Quartus Mapping netlist synthesized from the DC FPGA software. The default output netlist from the DC FPGA software is Verilog Quartus Mapping. The Verilog Quartus Mapping File format follows a subset of Verilog HDL rules. You can use the same Verilog Quartus Mapping netlist format with the Quartus II software and formal verification.

Use the `write` command to save your design work. The syntax for this command is shown in Example 11–6.

Example 11–6. Syntax Using the write Command

```
write -format <verilog|db|edif> -output <file name> <design list>
[-hierarchy] ↵
```

The `-hierarchy` option causes the DC FPGA software to write all the designs within the hierarchy of the current design. The DC FPGA default flow to interface with Quartus II software uses the Verilog Quartus Mapping netlist.

To generate a Verilog Quartus Mapping netlist, set the required settings using the commands shown in Example 11–7.

Example 11–7. Generating a Verilog Quartus Mapping Netlist

```
define_name_rules ALTERA -remove_internal_net_bus
change_names -rules ALTERA -hier
change_names –rules verilog –hier
write -format verilog -hier -o <design_top>.vqm
```

The Synopsys internal DB format is useful when you have synthesized your design and want to reuse it later in the DC FPGA software. The DB file contains your constraints and synthesized design netlist, and loads into the DC FPGA software faster than Verilog HDL or VHDL designs.

You can also write out your design constraints in Tcl format for export to the Quartus II software with the `write_par_constraint` command or by using the write_fpga command. These commands are explained in "Exporting Designs to the Quartus II Software".

Exporting Designs to the Quartus II Software

The DC FPGA software can create two Tcl scripts that start the Quartus II software, create your initial design project, apply the exported timing constraints, and compile your design in the Quartus II software.

You can generate the two Tcl scripts by using write and `write_par_constraint` command together, or by using the `write_fpga` command alone.

write_fpga Command

The recommended method to export all of the place-and-route files from the DC FPGA software is to use the `write_fpga` command. This command is used after the compile. Example 11–8 shows how the `write_fpga` command is used.

Example 11–8. Using the write_fpga Command after Compile

```
compile
write_fpga <outputdir>
```

The `write _fpga` command will do the following in one step:

Example 11–9. Using the write_fpga Command to Generate All Files

```
write -hier -f db -o $outputdir/top_module.db
write –hier –f edif –o $outputdir/top_module.edf
define_name_rules ALTERA -remove_internal_net_bus
change_names -rules ALTERA -hier
change_names –rules verilog –hier
write -format verilog -hier -o <design_top>.vqm
write_par_constraint $outputdir/top_module_quartus_setup.tcl
```

When you use the `write_fpga` command, it generates all files in the current work directory or in the directory you specify (entering an output directory is optional) and generates the output files based on the current design file name.

write and write_par_constraint Commands

The `write` command is used to generate a post synthesis netlist for place-and-route and formal verification. You should use a Verilog Quartus Mapping formatting netlist to work with the Quartus II software, beginning with the DC FPGA software, version 2005.09. Example 11–10 uses the write and write_par_constraint commands to generate the Verilog Quartus Mapping File and Tcl scripts:

Example 11–10. Using the write and write_par_constraint Commands

```
define_name_rules ALTERA -remove_internal_net_bus
change_names -rules ALTERA -hier
change_names –rules verilog –hier
write -format verilog -hier -o <design_top>.vqm
```

Tcl scripts that start the Quartus II software and forward annotate the timing constraints can be generated using the write_par_constraint command.

`write_par_constraint` <*user-specified file name*>`.tcl` ↵

This command generates both Tcl scripts in one operation. The first Tcl script has the name you specify in the `write_par_constraint` command. This script creates and compiles your Quartus II project. The second script is automatically generated and named <*top_module*>_const**.tcl** by default and contains your exported timing constraints from the DC FPGA software. This constraint file is sourced by the <*user-specified file name*>**.tcl** script and applies the timing constraints used in the DC FPGA software to your project in the Quartus II software.

For example, if your design is called **dma_controller**, and you run the command, `write_par_constraint run_quartus.tcl`, the DC FPGA software produces two Tcl scripts called `run_quartus.tcl` and `dma_controller_const.tcl`.

Using Tcl Scripts with Quartus II Software

To use this Tcl script in the Quartus II Tcl shell, type the following command at a command prompt:

`quartus_sh -t` <*user-specified file name*>`.tcl` ↵

To run this Tcl script in the Quartus II software GUI, type the following command at the Quartus II Tcl console prompt:

`source` <*user-specified file name*>`.tcl` ↵

The ability to run scripts in the Tcl console is useful when performing an initial compilation of your design to view post place-and-route timing and device utilization results, but the advanced Quartus II options that control the compilation process are not available.

To create a Quartus II project without performing compilation automatically, remove these lines from the script:

```
load_package flow
execute_flow -compile
```

Example 11–11. An Example Script

```
#############################################################################
# Generated by DC FPGA X-2005.09 on Wed Aug 10 04:20:01 2005
#
# Description: This TCL script is generated by DC FPGA using
#              write_par_constraint command. It is used to create a new Quartus
#              II project, specify timing constraint assignments in Quartus II,
#              and run quartus_map, quartus_fit, quartus_tan, & quartus_asm.
#
# Usage:  To execute this TCL script in batch mode: quartus_sh -t turboTop.tcl
#         To execute this TCL script in Quartus II GUI: source turboTop.tcl
#
#
#************    WARNING    **********    WARNING    **************************
#
# Please ensure the P&R netlist name is represented correctly in this tcl file.
# You may need to change the file_name variable to match your actual netlist
# name.
#
#############################################################################

# Set the file_name and  project_name variable
set file_name turboTop.vqm
set project_name turboTop

# Close the project if open
if [is_project_open] {
   project_close
}

# Create a new project
project_new -overwrite -family STRATIXII -part EP2S30F484C3 $project_name

# Make global assignments
set_global_assignment -name TOP_LEVEL_ENTITY $project_name

#############################################################################
# if you are using Verilog P&R netlist, please comment out EDIF assignment
# and uncomment the VERILOG assignment below.

#set_global_assignment -name EDIF_FILE $file_name
set_global_assignment -name  VQM_FILE $file_name
#############################################################################
```

```
set_global_assignment -name ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP ON
#set_global_assignment -name ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP OFF
set_global_assignment -name EDA_DESIGN_ENTRY_SYNTHESIS_TOOL -value "Design Compiler FPGA"
set_global_assignment -name EDA_INPUT_VCC_NAME -value VDD -section_id
eda_design_synthesis
set_global_assignment -name EDA_INPUT_GND_NAME -value GND -section_id
eda_design_synthesis
set_global_assignment -name EDA_LMF_FILE -value dc_fpga.lmf -section_id
eda_design_synthesis
set_global_assignment -name VERILOG_LMF_FILE dc_fpga.lmf
set_global_assignment -name FITTER_EFFORT "STANDARD FIT"

# Source in the design timing constraint file
source $project_name\_cons.tcl

# The following runs quartus_map, quartus_fit, quartus_tan, & quartus_asm
load_package flow
execute_flow -compile
project_close
```

After synthesis in the DC FPGA software, the technology-mapped design is written to the current project directory as an Verilog Quartus Mapping netlist file. The project configuration script (<*user-specified file name*>**.tcl**) is used to create and compile a Quartus II project containing your Verilog Quartus Mapping netlist. The example script makes basic project assignments such as assigning the target device as specified in the DC FPGA software. The project configuration script calls the place-and-route constraints script to make your timing constraints. The place-and-route constraints script (<*top module*>_const**.tcl**) forward-annotates the timing constraints that you made in the DC FPGA software, including false path assignments, multi-cycle assignments, timing groups, and related clocks. This integration means that you need to enter these constraints only once, in the DC FPGA software, and they are passed automatically to the Quartus II software.

Place and Route with the Quartus II Software

After you have created your Quartus II project and successfully loaded your Verilog Quartus Mapping netlist into the Quartus II project, you can use the Quartus II software to perform place-and-route. The Synopsys DC FPGA software uses only worst case timing delays and constraints, and does not optimize minimum timing requirements. Altera recommends that you add minimum timing constraints and perform minimum timing analysis in the Quartus II software.



For more information about these advance features, area optimization, and timing closure, refer to the *Quartus II Handbook*.

You can use the Quartus II software to obtain accurate prediction of post-conversion f_{MAX} performance and power consumption characteristics when migrating from a high-density FPGA to a cost-optimized, high-volume structured ASIC such as a HardCopy Stratix device.

The Quartus II software place-and-route algorithms can use register packing, register retiming, automatic logic duplication, and what-you-see-is-what-you-get (WYSIWYG) primitive re-synthesis technologies to increase logic utilization in your device and to deliver superior f_{MAX} performance at extremely high logic utilization.



For more information, refer to the *Quartus II Support for HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook.*

Formality Software Support

Beginning with version 4.2, the Quartus II software interfaces with the Formality software from Synopsys. Formality software verifies logic equivalency between the RTL and DC FPGA post-synthesis netlist, and between the DC FPGA post-synthesis netlist and the Quartus II post-place-and-route netlist. A synthesized verilog netlist generated by the DC FPGA is required to use with formality flow. Formality supports Stratix II, Stratix and Stratix GX device families.



For more information about how to set the required synthesis settings to generate a valid formal verification netlist and to use the Formality software for equivalence checking, refer to the *Synopsys Formality Support* chapter in volume 3 of the *Quartus II Handbook*.

Conclusion

Large FPGA designs require advanced synthesis of their HDL code. Taking advantage of the Synopsys DC FPGA software and the Quartus II software allows you to develop high-performance designs while occupying as little programmable logic resources as possible. The DC FPGA software and Quartus II software combination is an excellent solution for the high density designs using Altera FPGA devices.

Referenced Documents

This chapter references the following documents:

- *Quartus II Support for HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Synopsys DC FPGA User Guide*
- *Synopsys Formality Support* chapter in volume 3 of the *Quartus II Handbook*

Document Revision History

Table 11–3 shows the revision history for this chapter.

Table 11–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 11–26.	Updated for the Quartus II software version 7.2.
May 2007 v7.1.0	Added Referenced Documents.	—
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Added revision history to the chapter.	—
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.	—
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 11 was formerly chapter 13 in version 5.0.	—
May 2005 v5.0.0	Chapter 13 was formerly chapter 11 in version 4.2.	—
December 2004 v1.1	• Chapter 12 was formerly Chapter 13 in version 4.1. • Updated information. • New functionary for Quartus II software version 4.2. • Moved figure 12-3 within the chapter.	—
June 2004 v1.0	Initial release.	—

12. Analyzing Designs with Quartus II Netlist Viewers

QII51013-7.2.0

Introduction

As FPGA designs grow in size and complexity, the ability to analyze how your synthesis tool interprets your design becomes critical. Often, with today's advanced designs, several design engineers are involved in coding and synthesizing different design blocks, making it difficult to analyze and debug the design. The Quartus® II RTL Viewer, State Machine Viewer, and Technology Map Viewer provide powerful ways to view your initial and fully mapped synthesis results during the debugging, optimization, or constraint entry process.

The first section in this chapter, "When to Use Viewers: Analyzing Design Problems", describes examples of using the viewers to analyze your design at various stages of the design cycle. The sections following this provide an introduction to the Quartus II design flow using the netlist viewers, an overview of each viewer, and an explanation of the user interface. These sections describe the following tasks:

- How to navigate and filter schematics
- How to probe to and from other windows within the Quartus II software
- How to view a timing path from the Timing Analyzer report

This chapter contains the following sections regarding the netlist viewers:

- "Introduction to the User Interface" on page 12–7
- "Navigating the Schematic View" on page 12–21
- "Filtering in the Schematic View" on page 12–34
- "Probing to Source Design File and Other Quartus II Windows" on page 12–42
- "Probing to the Viewers from Other Quartus II Windows" on page 12–44
- "Viewing a Timing Path" on page 12–45
- "Other Features in the Schematic Viewer" on page 12–47

The final section, "Debugging HDL Code with the State Machine Viewer", provides a detailed example that uses the viewer to analyze a design and quickly resolve a design problem.

When to Use Viewers: Analyzing Design Problems

You can use the netlist viewers to analyze your design to determine how it was interpreted by the Quartus II software. This section provides simple examples of how to use the RTL viewers, State Machine, and Technology Map Viewers to analyze problems encountered in the design process.

The following sections contain information about how the netlist viewers display your design:

- "Quartus II Design Flow with Netlist Viewers"
- "RTL Viewer Overview"
- "State Machine Viewer Overview"
- "Technology Map Viewer Overview"

Using the RTL Viewer is a good way to view your initial synthesis results to determine whether you have created the desired logic, and that the logic and connections have been interpreted correctly by the software. You can use the RTL Viewer and the State Machine Viewer to visually check your design before simulation or other verification processes. Catching design errors at this early stage of the design process can save you valuable time.

If you see unexpected behavior during verification, you can use the RTL Viewer to trace through the netlist and ensure that the connections and the logic in your design are as expected. You can also use the State Machine Viewer to view state machine transitions and transition equations. Viewing the design can help you find and analyze the source of design problems. If your design looks correct in the RTL Viewer, you know to focus your analysis on later stages of the design process and investigate potential timing violations or issues in the verification flow itself.

You can use the Technology Map Viewer to look at the results at the end of synthesis and technology mapping by running the viewer after performing Analysis and Synthesis. If you have compiled your design through the Fitter stage, you can view your post-mapping netlist in the Technology Map Viewer (Post-Mapping), and your post-fitting netlist in the Technology Map Viewer. If you perform only Analysis and Synthesis, both viewers display the same post-mapping netlist.

In addition, you can use the RTL Viewer or Technology Map Viewer to locate the source of a particular signal, which can help you debug your design. Use the navigation techniques described in this chapter to search easily through the design. You can trace back from a point of interest to find the source of the signal and ensure the connections are as expected.

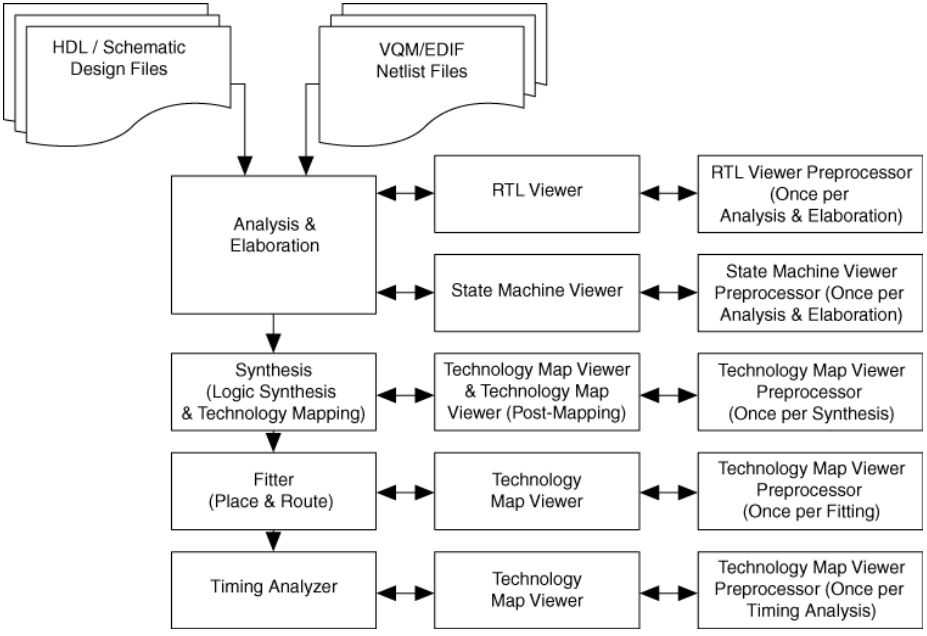
You can also use the Technology Map Viewer to help you locate post-synthesis nodes in your netlist and make assignments when optimizing your design. This functionality is useful, for example, when making a multicycle clock timing assignment between two registers in your design. Start at an I/O port and trace forward or backward through the design and through levels of hierarchy to find nodes that interest you, or locate a specific register by visually inspecting the schematic.

The RTL Viewer, State Machine Viewer, and Technology Map Viewer can be used in many other ways throughout the design, debugging, and optimization stages. Viewing the design netlist is a powerful way to analyze design problems. This chapter shows how you can use the various features of the netlist viewers to increase your productivity when analyzing a design.

Quartus II Design Flow with Netlist Viewers

The first time you open one of the netlist viewers after compiling the design, a preprocessor stage runs automatically before the viewer opens. If you close the viewer and open it again later without recompiling the design, the viewer opens immediately without performing the preprocessing stage. Figure 12–1 shows how the netlist viewers fit into the basic Quartus II design flow.

Figure 12–1. Quartus II Design Flow Including the RTL Viewer and Technology Map Viewer



Each viewer requires that your design has been compiled with the minimum compilation stage listed below before the viewer can run the preprocessor and open the design.

- To open the RTL Viewer or State Machine Viewer, you must first perform Analysis and Elaboration.
- To open the Technology Map Viewer or the Technology Map Viewer (Post-Mapping), you must first perform Analysis and Synthesis.



If you open one of the viewers without first compiling the design with the appropriate minimum compilation stage, the viewer does not appear. Instead, the Quartus II software issues an error message instructing you to run the necessary compilation stage and restart the viewer.

Both viewers display the results of the last successful compilation. Therefore, if you make a design change that causes an error during Analysis and Elaboration, you cannot view the netlist for the new design files, but you can still see the results from the last successfully compiled version of the design files. If you receive an error during compilation and you have not yet successfully run the appropriate compilation stage for your project, the viewer cannot be displayed; in this case, the Quartus II software issues an error message when you try to open the viewer.



If the viewer window is open when you start a new compilation, the viewer closes automatically. You must open the viewer again to view the new design netlist after compilation completes successfully.

RTL Viewer Overview

The Quartus II RTL Viewer allows you to view a register transfer level (RTL) graphical representation of your Quartus II integrated synthesis results or your third-party netlist file within the Quartus II software.

You can view results after Analysis and Elaboration when your design uses any supported Quartus II design entry method, including Verilog HDL Design Files (**.v**), SystemVerilog Design Files (**.sv**), VHDL Design Files (**.vhd**), AHDL Text Design Files (**.tdf**), schematic Block Design Files (**.bdf),** or schematic Graphic Design Files (**.gdf)** imported from the MAX+PLUS® II software. You can also view the hierarchy of atom primitives (such as device logic cells and I/O ports) when your design uses a synthesis tool to generate a Verilog Quartus Mapping File (**.vqm**) or Electronic Design Interchange Format (**.edf**) netlist file. Refer to Figure 12–1 for a flow diagram.

The Quartus II RTL Viewer displays a schematic view of the design netlist after analysis and elaboration or netlist extraction is performed by the Quartus II software, but before technology mapping and any synthesis or fitter optimization algorithms occur. This view is not the final design structure because optimizations have not yet occurred. This view most closely represents your original source design. If you synthesized your design using the Quartus II integrated synthesis, this view shows how the Quartus II software interpreted your design files. If you are using a third-party synthesis tool, this view shows the netlist written by your synthesis tool.

When displaying your design, the RTL Viewer optimizes the netlist to maximize readability in the following ways:

- Logic with no fan-out (its outputs are unconnected) and logic with no fan-in (its inputs are unconnected) are removed from the display.
- Default connections such as `VCC` and `GND` are not shown.
- Pins, nets, wires, module ports, and certain logic are grouped into buses where appropriate.
- Constant bus connections are grouped.
- Values are displayed in hexadecimal format.
- `NOT` gates are converted to bubble inversion symbols in the schematic.
- Chains of equivalent combinational gates are merged into a single gate. For example, a 2-input `AND` gate feeding a 2-input `AND` gate is converted to a single 3-input `AND` gate.
- State machine logic is converted into a state diagram, state transition table, and state encoding table, which are displayed in the State Machine Viewer.

To run the RTL Viewer for a Quartus II project, first analyze the design to generate an RTL netlist. To analyze the design and generate an RTL netlist, on the Processing menu, point to Start and click **Start Analysis & Elaboration**. You can also perform a full compilation on any process that includes the initial Analysis and Elaboration stage of the Quartus II compilation flow.

To run the viewer, on the Tools menu, point to Netlist Viewers and click **RTL Viewer**, or select **RTL Viewer** from the **Applications** toolbar.



By default, the **Applications** toolbar does not display in the Quartus II user interface. To add the toolbar, on the Tools menu, click **Customize**. On the **Customize** dialog box, click the **Toolbars** tab under **Toolbars**, and turn on **Applications**. Click **Close**.

You can set the RTL Viewer preprocessing to run during a full compilation, which means you can launch the RTL Viewer after Analysis and Synthesis has completed, but while the Fitter is still running. In this case, you do not have to wait for the Fitter to finish before viewing the schematic. This technique is useful for a large design that requires a substantial amount of time in the place-and-route stage.

To set the RTL Viewer preprocessing to run during compilation, on the Assignments menu, click **Settings**. In the **Category** list, select Compilation Process Settings and turn on **Run RTL Viewer preprocessing during compilation**. By default, this option is turned off.

State Machine Viewer Overview

The State Machine Viewer presents a high-level view of finite state machines in your design. The State Machine Viewer provides a graphical representation of the states and their related transitions, as well as a state transition table that displays the condition equation for each of the state transitions, and encoding information for each state.

To run the State Machine Viewer, on the Tools menu, point to Netlist Viewers and click **State Machine Viewer**. To open the State Machine Viewer for a particular state machine, double-click the state machine instance in the RTL Viewer, or right-click the state machine instance, and click **Hierarchy Down**.

Technology Map Viewer Overview

The Quartus II Technology Map Viewer provides a technology-specific, graphical representation of your design after Analysis and Synthesis or after the Fitter has mapped your design into the target device. The Technology Map Viewer shows the hierarchy of atom primitives (such as device logic cells and I/O ports) in your design. For supported families, you can also view the internal registers and look-up tables (LUTs) inside logic cells (LCELLs) and registers in I/O atom primitives. Refer to "Viewing Contents of Atom Primitives in the Technology Map Viewer" on page 12–22 for details.



Where possible, the port names of each hierarchy are maintained throughout synthesis. However, port names may change or be removed from the design. For example, if a port is unconnected or driven by `GND` or `VCC`, it is removed during synthesis. When a port name is changed, the port is assigned a related user logic name in the design, or a generic port name such as `IN1` or `OUT1`.

You can view your Quartus II technology-mapped results after synthesis, fitting, or timing analysis. To run the Technology Map Viewer for a Quartus II project, on the Processing menu, point to Start and click **Start Analysis & Synthesis** to synthesize and map the design to the target

technology. At this stage, the Technology Map Viewer shows the same post-mapping netlist as does the Technology Map Viewer (Post-Mapping). You can also perform a full compilation, or any process that includes the synthesis stage in the compilation flow.

If you have completed the Fitter stage, the Technology Map Viewer shows the changes made to your netlist by the Fitter, such as physical synthesis optimizations, while the Technology Map Viewer (Post-Mapping) shows the post-mapping netlist. If you have completed the Timing Analysis stage, you can locate timing paths from the Timing Analyzer report in the Technology Map Viewer (refer to "Viewing a Timing Path" on page 12–45 for details). Refer to Figure 12–1 on page 12–3 for a flow diagram.

To run the Technology Map Viewer, on the Tools menu, point to Netlist Viewers and click **Technology Map Viewer**, or select **Technology Map Viewer** from the **Applications** toolbar.

To run the Technology Map Viewer (Post-Mapping), on the Tools menu, point to Netlist Viewers and click **Technology Map Viewer (Post-Mapping)**.

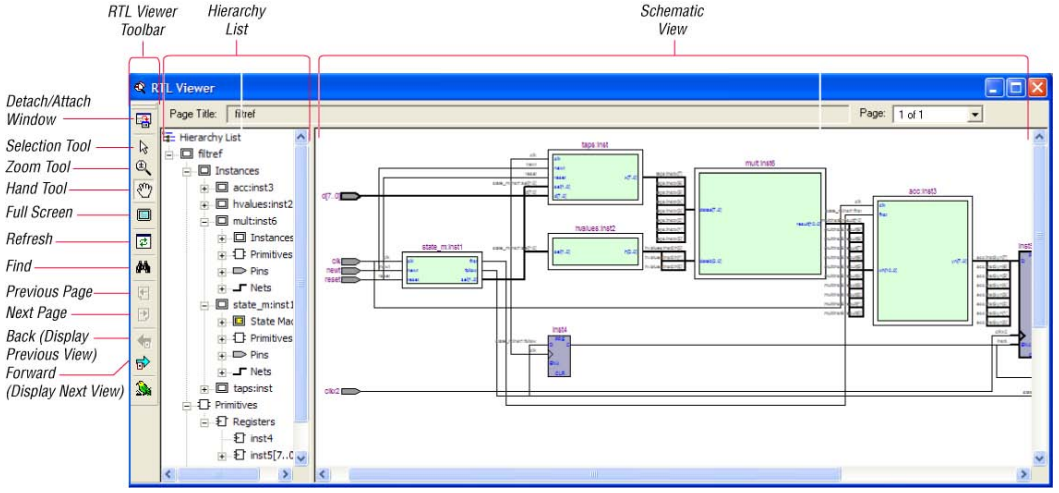
Introduction to the User Interface

The RTL Viewer window and Technology Map Viewer window each consist of two main parts: the schematic view and the hierarchy list. Figure 12–2 shows the RTL Viewer window and indicates these two parts. Both viewers also contain a toolbar that gives you tools to use in the schematic view.

You can have only one RTL Viewer, one Technology Map Viewer, and one State Machine Viewer window open at a time, although each window can show multiple pages. The window for each viewer has characteristics similar to other "child" windows in the Quartus II software; it can be resized and moved, minimized or maximized, tiled or cascaded, and moved in front of or behind other windows.

You can detach the window and move it outside the Quartus II main interface. To detach a window, click the **Detach Window** icon on the toolbar, or, on the Window menu, click **Detach Window**. To attach the detached window back to the Quartus II main interface, click the **Attach Window** icon on the toolbar, or, on the Window menu, click **Attach Window**.

Figure 12–2. RTL Viewer Window and RTL Toolbar



Schematic View

The schematic view is shown on the right side of the RTL Viewer and Technology Map Viewer. It contains a schematic representing the design logic in the netlist. This view is the main screen for viewing your gate-level netlist in the RTL Viewer and your technology-mapped netlist in the Technology Map Viewer.

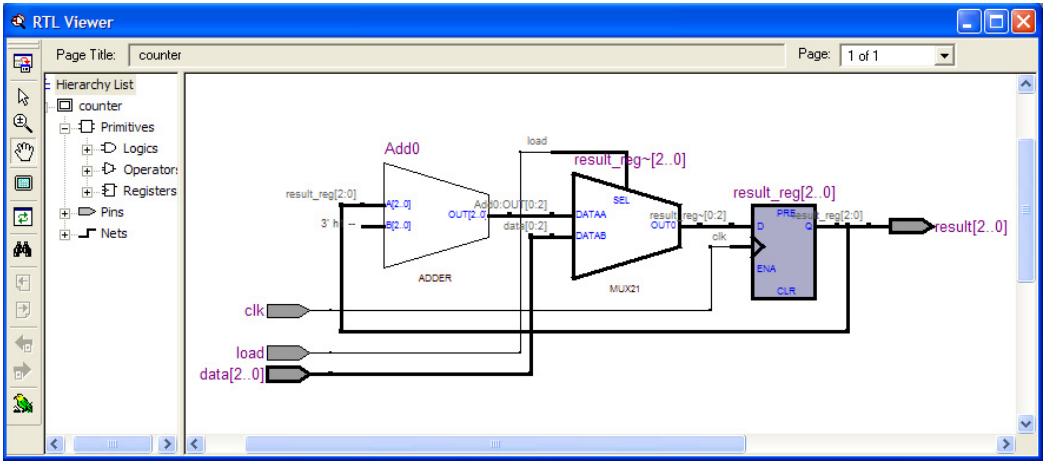
Schematic Symbols

The symbols for nodes in the schematic represent elements of your design netlist. These elements include input and output ports, registers, logic gates, Altera® primitives, high-level operators, and hierarchical instances.

Figure 12–3 shows an example of an RTL Viewer schematic for a 3-bit synchronous loadable counter. Example 12–1 shows the Verilog HDL code that produced this schematic. This example includes multiplexers and a group of registers (Table 12–1 on page 12–10) in a bus along with an ADDER operator (Table 12–3 on page 12–13) inferred by the counting function in the HDL code.

The schematic in Figure 12–3 displays wire connections between nodes with a thin black line, and bus connections with a thick black line.

Figure 12–3. Example Schematic Diagram in the RTL Viewer



Example 12–1. Code Sample for Counter Schematic Shown in Figure 12–3

```
module counter (input [2:0] data, input clk, input load, output [2:0] result);
    reg [2:0] result_reg;
    always @ (posedge clk)
        if (load)
            result_reg <= data;
        else
            result_reg <= result_reg + 1;
    assign result = result_reg;
endmodule
```

Figure 12–4 shows a portion of the corresponding Technology Map Viewer schematic with a compiled design that targets a Stratix® device. In this schematic, you can see the LCELL (logic cell) device-specific primitives that represent the counter function, labeled with their post-synthesis node names. The REGOUT port represents the output of the register in the LCELL, and the COMBOUT port represents the output of the combinational logic in the LUT of the LCELL. The hexadecimal number in parentheses below each LCELL primitive represents the LUT mask, which is a hexadecimal representation of the logic function of the LCELL.

Figure 12–4. Example Schematic Diagram in the Technology Map Viewer

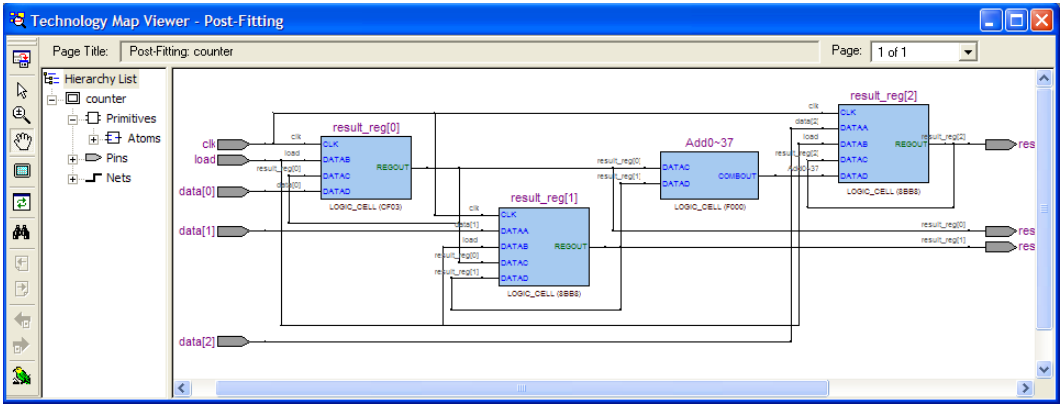


Table 12–1 lists and describes the primitives and basic symbols that you can display in the schematic view of the RTL Viewer and Technology Map Viewer. Table 12–3 on page 12–13 lists and describes the additional higher level operator symbols used in the RTL Viewer schematic view.



The logic gates and operator primitives appear only in the RTL Viewer. Logic in the Technology Map Viewer is represented by atom primitives such as registers and LCELLs.

Table 12–1. Symbols in the Schematic View (Part 1 of 3)

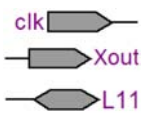
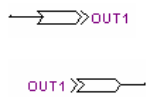
Symbol	Description
I/O Ports 	An input, output, or bidirectional port in the current level of hierarchy. A device input, output, or bidirectional pin when viewing the top-level hierarchy. The symbol can represent a bus. Only one wire is connected to the bidirectional symbol, representing both the input and the output paths. Input symbols appear on the left-most side of the schematic. Output and bidirectional symbols appear on the right-most side of the schematic.
I/O Connectors 	An input or output connector, representing a net that comes from another page of the same hierarchy (refer to "Partitioning the Schematic into Pages" on page 12–28). To go to the page that contains the source or the destination, right-click on the net and choose the page from the menu (refer to "Following Nets Across Schematic Pages" on page 12–29).
Hierarchy Port Connector 	A connector representing a port relationship between two different hierarchies. A connector indicates that a path passes through a port connector in a different level of hierarchy.

Table 12–1. Symbols in the Schematic View (Part 2 of 3)

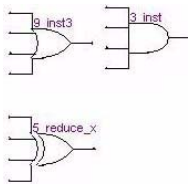
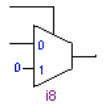
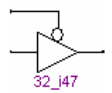
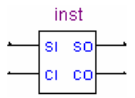
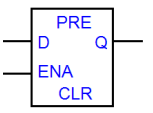
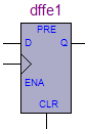
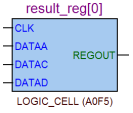
Symbol	Description
OR, AND, XOR Gates 	An OR, AND, or XOR gate primitive (the number of ports can vary). A small circle (bubble symbol) on an input or output port indicates the port is inverted.
MUX 	A multiplexer (MUX) primitive with a selector port that selects between port 0 and port 1. A MUX with more than two inputs is displayed as an operator (refer to "Operator Symbols in the RTL Viewer Schematic View" on page 12–13).
BUFFER 	A buffer primitive. The figure shows the tri-state buffer, with an inverted output enable port. Other buffers without an enable port include LCELL, SOFT, CARRY, and GLOBAL. The NOT gate and EXP expander buffers use this symbol without an enable port and with an inverted output port.
CARRY_SUM 	A CARRY_SUM buffer primitive with the following ports: ● SI – SUM IN ● SO – SUM OUT ● CI – CARRY IN ● CO – CARRY OUT
LATCH 	A latch primitive with the following ports: ● D – data input ● ENA – enable input ● Q – data output ● PRE – preset ● CLR – clear
DFFE/DFFEA/DFFAES 	A DFFE (data flipflop with enable) primitive, with the same ports as a latch and a clock trigger. The other flipflop primitives are similar: ● DFFEA (data flipflop with enable and asynchronous load) primitive with additional ALOAD asynchronous load and ADATA data signals ● DFFEAS (data flipflop with enable and both synchronous and asynchronous load), which has ASDATA as the secondary data port
Atom Primitive 	Primitives are low-level nodes that cannot be expanded to any lower hierarchy. The symbol displays the port names, the primitive type, and its name. The blue shading indicates an atom primitive in the Technology Map Viewer that allows you to view the internal details of the primitive. Refer to "Viewing Contents of Atom Primitives in the Technology Map Viewer" on page 12–22 for details.

Table 12–1. Symbols in the Schematic View (Part 3 of 3)

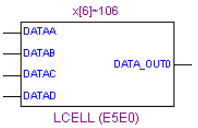
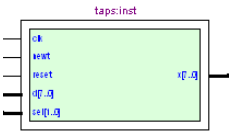
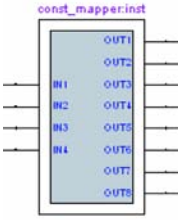
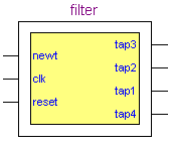
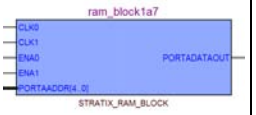
Symbol	Description
Other Primitive 	Any primitive that does not fall into the categories above. Primitives are low-level nodes that cannot be expanded to any lower hierarchy. The symbol displays the port names, the primitive or operator type, and its name. The figure shows an LCELL WYSIWYG primitive, with DATAA to DATAD and COMBOUT port connections. This type of LCELL primitive would be found in the Technology Map Viewer for technology-specific atom primitives when the contents of the atom primitive cannot be viewed. The RTL Viewer contains similar primitives if the source design was a VQM or EDIF netlist.
Instance 	An instance in the design that does not correspond to a primitive or operator (generally a user-defined hierarchy block), indicated by the double outline and green shading. The symbol displays the instance name. To open the schematic for the lower level hierarchy, right-click and choose the appropriate command (refer to "Traversing and Viewing the Design Hierarchy" on page 12–21).
Encrypted Instance 	A user-defined encrypted instance in the design, indicated by the double outline and gray shading. The symbol displays the instance name. You cannot open the schematic for the lower level hierarchy, because the source design is encrypted.
State Machine Instance 	A finite state machine instance in the design, indicated by the double outline and yellow shading. Double-clicking this instance opens the State Machine Viewer. Refer to "State Machine Viewer" on page 12–18 for more details.
RAM 	A synchronous memory instance with registered inputs and optionally registered outputs, indicated by purple shading. The symbol shows the device family and the type of TriMatrix memory block. This figure shows a true dual-port memory block in a Stratix M-RAM block.
Logic Cloud 	A logic cloud is a group of combinational logic, indicated by a cloud symbol. Refer to "Grouping Combinational Logic into Logic Clouds" on page 12–32 for more details.

Table 12–2 lists and describes the symbol used only in the State Machine Viewer.

Table 12–2. Symbol Available Only in the State Machine Viewer

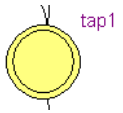
Symbol	Description
State Node 	The node representing a state in a finite state machine. State transitions are indicated with arcs between state nodes. The double circle border indicates the state connects to logic outside the state machine, while a single circle border indicates the state node does not feed outside logic.

Table 12–3 lists and describes the additional higher level operator symbols used in the RTL Viewer schematic view.

Table 12–3. Operator Symbols in the RTL Viewer Schematic View (Part 1 of 2)

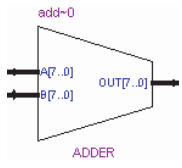
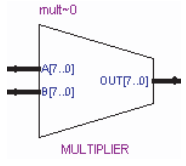
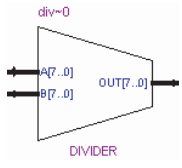
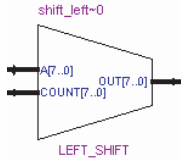
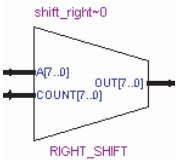
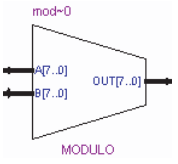
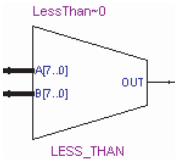
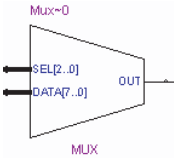
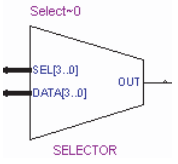
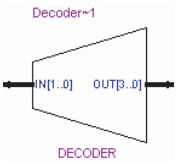
Symbol	Description
	An adder operator: `OUT = A + B`
	A multiplier operator: `OUT = A × B`
	A divider operator: `OUT = A / B`
	A left shift operator: `OUT = (A << COUNT)`

Table 12–3. Operator Symbols in the RTL Viewer Schematic View (Part 2 of 2)

Symbol	Description
	A right shift operator: `OUT = (A >> COUNT)`
	A modulo operator: `OUT = (A % B)`
	A less than comparator: `OUT = (A <= B : A > B)`
	A multiplexer: `OUT = DATA [SEL]` The data range size is $2^{\text{sel range size}}$
	A selector: A multiplexer with one-hot select input and more than two input signals
	A binary number decoder: `OUT` = (binary_number (`IN`) == x) for x = 0 to x = $2^{(n+1)}$ - 1

Selecting an Item in the Schematic View

To select an item in the schematic view, ensure that the **Selection Tool** is enabled in the viewer toolbar (this tool is enabled by default). Click on an item in the schematic view to highlight it in red.

Select multiple items by pressing the Shift or Ctrl key while selecting with your mouse. You can also select all nodes in a region by selecting a rectangular box area with your mouse cursor when the **Selection Tool** is enabled. To select nodes in a box, move your mouse to one corner of the area you want to select, click the mouse button, and drag the mouse to the opposite corner of the box, then release the mouse button. By default, creating a box like this highlights and selects all nodes in the selected area (instances, primitives, and pins), but not the nets. The **Viewer Options** dialog box provides an option to select nets. To include nets, right-click in the schematic and click **Viewer Options**. In the **Net Selection** section, turn on the **Select entire net when segment is selected** option.

Items selected in the schematic view are automatically selected in the hierarchy list (refer to the "Hierarchy List" on page 12–16). The list expands automatically if required to show the selected entry. However, the list does not collapse automatically when entries are not being used or are deselected.

When you select a hierarchy box, node, or port in the schematic view, the item is highlighted in red but none of the connecting nets are highlighted. When you select a net (wire or bus) in the schematic view, all connected nets are highlighted in red. The selected nets are highlighted across all hierarchy levels and pages. Net selection can be useful when navigating a netlist because you see the net highlighted when you traverse between hierarchy levels or pages.

In some cases, when you select a net that connects to nets in other levels of the hierarchy, these connected nets also are highlighted in the current hierarchy. If you prefer that these nets not be highlighted, use the **Viewer Options** dialog box option to highlight a net only if the net is in the current hierarchy. Right-click in the schematic and click **Viewer Options**. In the **Net Selection** section, turn on the **Limit selections to current hierarchy** option.

Moving and Panning in the Schematic View

When the schematic view page is larger than the portion currently displayed, you can use the scroll bars at the bottom and right side of the schematic view to see other areas of the page.

You can also use the Hand Tool to "grab" the schematic page and drag it in any direction. Enable the Hand Tool with the toolbar button. Click and drag to move around the schematic view without using the scroll bars.

In addition to the scroll bars and Hand Tool, you can use the middle-mouse/wheel button to move and pan in the schematic view. Click the middle-mouse/wheel button once to enable the feature. Move the mouse or scroll the wheel to move around the schematic view. Click the middle-mouse/wheel button again to turn the feature off.

Hierarchy List

The hierarchy list is displayed on the left side of the viewer window. The hierarchy list displays the entire netlist in a tree format based on the hierarchical levels of the design. Within each level, similar elements are grouped into sub-categories. Using the hierarchy list, you can traverse through the design hierarchy to view the logic schematic for each level. You can also select an element in the hierarchy list to be highlighted in the schematic view.



Nodes inside atom primitives are not listed in the hierarchy list.

For each module in the design hierarchy, the hierarchy list displays the applicable elements listed in Table 12–4. Click the + icon to expand an element.

Table 12–4. Hierarchy List Elements

Elements	Description
Instances	Modules or instances in the design that can be expanded to lower hierarchy levels.
State Machines	State machine instances in the design that can be viewed in the State Machine Viewer.
Primitives	Low-level nodes that cannot be expanded to any lower hierarchy level. These include: • Registers and gates that you can view in the RTL Viewer when using Quartus II integrated synthesis • Logic cell atoms in the Technology Map Viewer or in the RTL Viewer when using a VQM or EDIF from third-party synthesis software In the Technology Map Viewer, you can view the internal implementation of certain atom primitives, but you can not traverse into a lower level of hierarchy.
Pins	The I/O ports in the current level of hierarchy. • Pins are device I/O pins when viewing the top hierarchy level, and are I/O ports of the design when viewing the lower levels. • When a pin represents a bus or an array of pins, expand the pin entry in the list view to see individual pin names.
Nets	Nets or wires connecting the nodes. When a net represents a bus or array of nets, expand the net entry in the tree to see individual net names.
Logic Clouds	A group of related combinational logics of a particular source. You can automatically or manually group combinational logics or ungroup logic clouds in your design.

Selecting an Item in the Hierarchy List

When you click any item in the hierarchy list, the viewer performs the following actions:

- Searches for the item in the currently viewed pages, and displays the page containing the selected item in the schematic view if it is not currently displayed. (If you are currently viewing a filtered netlist, for example, the relevant page within the filtered netlist is displayed.)
- If the selected item is not found in the currently viewed pages, the entire design netlist is searched, and the item is displayed in a default view.
- Highlights the selected item in red in the schematic view.

When you double-click an instance in the hierarchy list, the viewer displays the underlying implementation of the instance.

You can select multiple items by pressing the Shift or Ctrl key while selecting with your mouse. When you right-click an item in the hierarchy list, you can navigate in the schematic view using the **Filter** and **Locate** commands. Refer to "Filtering in the Schematic View" on page 12–34 and "Probing to Source Design File and Other Quartus II Windows" on page 12–42 for more information.

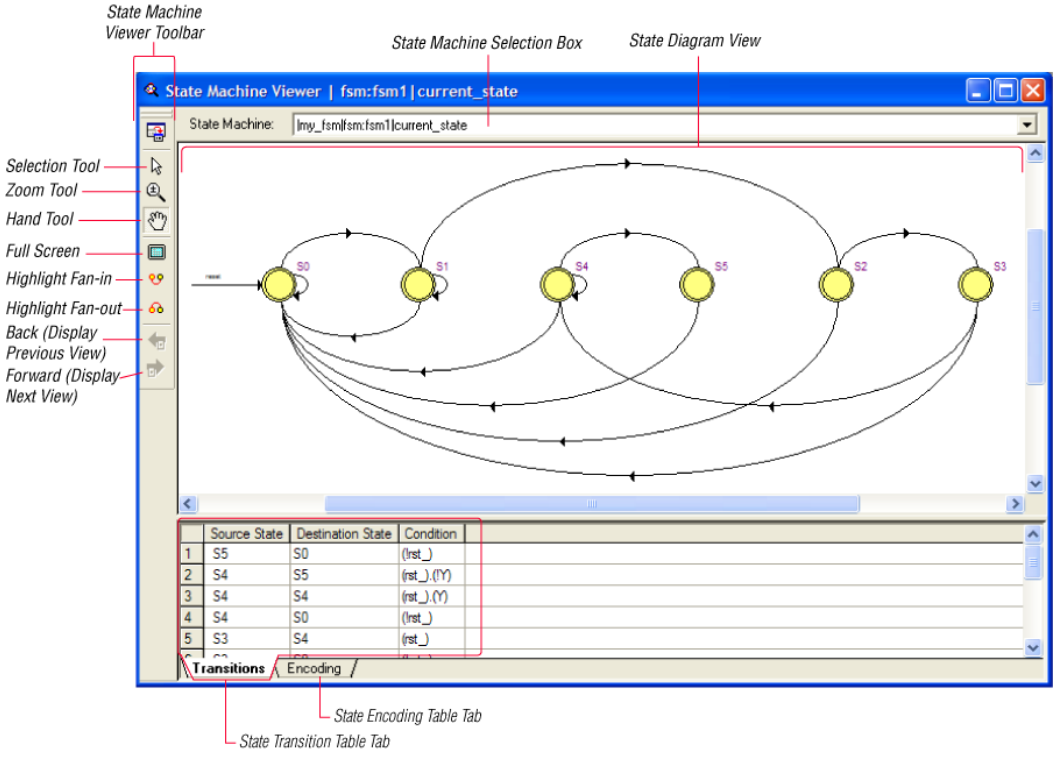
State Machine Viewer

The State Machine Viewer displays a graphical representation of the state machines in your design. You can open the State Machine Viewer in any of the following ways:

- On the Tools menu, point to Netlist Viewers, and click **State Machine Viewer**
- Double-click on a state machine instance in the RTL Viewer
- Right-click on a state machine instance in the RTL Viewer, and click **Hierarchy Down**
- Select a state machine instance in the RTL Viewer, and on the Project menu, point to Hierarchy and click **Down**

Figure 12–5 shows an example of the State Machine Viewer for a simple state machine. The State Machine toolbar on the left side of the viewer provides tools you can use in the state diagram view.

Figure 12–5. State Machine in the State Machine Viewer



State Diagram View

The state diagram view is shown at the top of the State Machine Viewer window. It contains a diagram of the states and state transitions.

The nodes that represent each state are arranged horizontally in the state diagram view with the initial state (the state node that receives the reset signal) in the left-most position. Nodes that connect to logic outside of the state machine instance are represented by a double circle. The state transition is represented by an arc with an arrow pointing in the direction of the transition.

When you select a node in the state diagram view, if you turn on the **Highlight Fan-in** or **Highlight Fan-out** command from the View menu or the State Machine Viewer toolbar, the respective fan-in or fan-out transitions from the node are highlighted in red.



An encrypted block with a state machine displays encoding information in the state encoding table, but does not display a state transition diagram or table.

State Transition Table

The state transition table on the **Transitions** tab at the bottom of the State Machine Viewer window displays the condition equation for each state transition. Each transition (each arc in the state diagram view) is represented by a row in the table. The table has the following three columns:

- **Source State**—the name of the source state for the transition
- **Destination State**—the name of the destination state for the transition
- **Condition**—the condition equation that causes the transition from source state to destination state

To see all of the transitions to and from each state name, click the appropriate column heading to sort on that column.

The text in each column is left-aligned by default; to change the alignment and more easily see the relevant part of the text, right-click in the column and click **Align Right**. To change back to left alignment, click **Align Left**.

You can click in any cell in the table to select it. To select all cells, right-click in the cell and click **Select All**; or, on the Edit menu, click **Select All**. To copy selected cells to the clipboard, right-click the cells and click **Copy Table**; or, on the Edit menu, point to Copy and click **Copy Table**. You can paste the table into any text editor as tab-separated columns.

State Encoding Table

The state encoding table on the **Encoding** tab at the bottom of the State Machine Viewer window displays the encoding information for each state transition.

To view state encoding information in the State Machine Viewer, you must have synthesized your design using **Start Analysis & Synthesis**. If you have only elaborated your design using **Start Analysis & Elaboration**, the encoding information is not displayed.

Selecting an Item in the State Machine Viewer

You can select and highlight each state node and transition in the State Machine Viewer. To select a state transition, click the arc that represents the transition.

When you select a state node, transition arc, or both in the state diagram view, the matching state node and equation conditions in the state transition table are highlighted. Conversely, when you select a state node, equation condition, or both in the state transition table, the corresponding state node and transition arc are highlighted in the state diagram view.

Switching Between State Machines

A design may contain multiple state machines. To choose which state machine to view, use the **State Machine** selection box located at the top of the State Machine Viewer. Click in the drop-down box and select the desired state machine.

Navigating the Schematic View

The previous sections provided an overview of the user interface for each netlist viewer, and how to select an item in each viewer. This section describes methods to navigate through the pages and hierarchy levels in the schematic view of the RTL Viewer and Technology Map Viewer.

Traversing and Viewing the Design Hierarchy

You can open different hierarchy levels in the schematic view using the hierarchy list (refer to "Hierarchy List" on page 12–16), or the **Hierarchy Up** and **Hierarchy Down** commands in the schematic view.

Use the **Hierarchy Down** command to go down into, or expand an instance's hierarchy, and open a lower level schematic showing the internal logic of the instance. Use the **Hierarchy Up** command to go up in hierarchy, or collapse a lower level hierarchy, and open the parent higher level hierarchy. When the **Selection Tool** is selected, the appropriate option is available when your mouse pointer is located over an area of the schematic view that has a corresponding lower or higher level hierarchy.

The mouse pointer changes as it moves over different areas of the schematic to indicate whether you can move up, down, or both up and down in the hierarchy (Figure 12–6). To open the next hierarchy level, right-click in that area of the schematic, and click **Hierarchy Down** or **Hierarchy Up**, as appropriate, or double-click in that area of the schematic.

Figure 12–6. Mouse Pointers Indicate How to Traverse Hierarchy



Flattening the Design Hierarchy

You can flatten the design hierarchy to view the design without hierarchical boundaries. To flatten the hierarchy from the current level and all the lower level hierarchies of the current design hierarchy, right-click in the schematic and click **Flatten Netlist**. To flatten the entire design, choose **Flatten Netlist** from the top-level schematic of the design.

Viewing the Contents of a Design Hierarchy within the Current Schematic

You can use the **Display Content** and **Hide Content** commands to show or hide a lower hierarchy level for a specific instance within the schematic for the current hierarchy level.

To display the lower hierarchy netlist of an instance on the same schematic as the remaining logic in the currently viewed netlist, right-click the selected instance and click **Display Content**.

To hide all of the lower hierarchy logic of a hierarchy box into a closed instance, right-click the selected instance and click **Hide Content**.

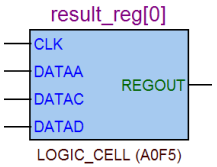
Viewing Contents of Atom Primitives in the Technology Map Viewer

In the Technology Map Viewer, you can view the contents of certain device atom primitives to see their underlying implementation details. For logic cell (`LCELL`) atoms in the Stratix and Cyclone® series of devices and in MAX® II devices, you can view the LUTs, registers, and logic gates. For I/O atoms in the Stratix and Cyclone series of devices, and HardCopy® II devices, you can view the registers and logic gates.

In addition, you can view the implementation of RAM and DSP blocks in certain devices. You can view the implementation of RAM blocks in the Stratix and Cyclone series of devices. You can view the implementation of DSP blocks only in the Stratix series of devices.

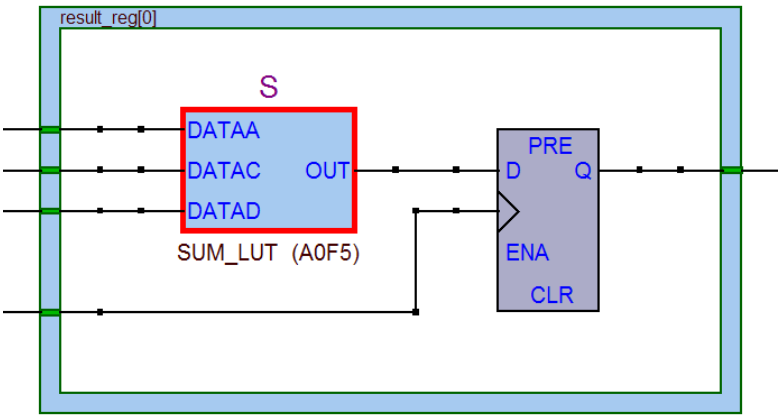
If you can view the contents of an atom instance, it is blue in the schematic view (Figure 12–7).

Figure 12–7. Instance That Can Be Expanded to View Internal Contents



To view the contents of one or more atom primitive instances, select the desired atom instances. Right-click a selected instance and click **Display Content**. You can also double-click on the desired atom instance to view the contents. Figure 12–8 shows an expanded version of the instance in Figure 12–7.

Figure 12–8. Internal Contents of the Atom Instance in Figure 12–7.



To hide the contents (and revert to the compact format), select and right-click the atom instance(s), and click **Hide Content**.

In the schematic view, the internal details within an atom instance can not be selected as individual nodes. Any mouse action on any of the internal details is treated as a mouse action on the atom instance.

Viewing the Properties of Instances and Primitives

You can view the properties of an instance or a primitive using the Properties dialog box. To view the properties of an instance or a primitive in the RTL Viewer or the Technology Map Viewer, right-click the node and click **Properties**.

The **Properties** dialog box contains the following information about the selected node:

- The parameter values of an instance.
- The active level of the port (for example, active high or active low). An active low port is denoted with an exclamation mark "!".
- The port's constant value (for example, VCC or GND). Table 12–5 describes the possible value of a port.

Table 12–5. Possible Port Values

Value	Description
VCC	The port is not connected and has VCC value (tied to VCC)
GND	The port is not connected and has GND value (tied to GND)
--	The port is connected and has value (other than VCC or GND)
Unconnected	The port is not connected and has no value (hanging)

In the look-up-table (LUT) of a logic cell (LCELL), the **Properties** dialog box contains the following additional information:

- The schematic of the LCELL.
- The Truth Table representation of the LCELL.
- The Karnaugh map representation of the LCELL.

Viewing LUT Representations in the Technology Map Viewer

You can view different representations of an LUT by right-clicking on the selected LUT, and selecting **Properties**. This feature is only applicable for the Stratix and Cyclone series of devices, and in MAX II devices. There are three tabs in the **Properties** dialog box, which you can choose from to view the LUT representations. The **Schematic** tab (see Figure 12–9) shows you the equivalent gate representations of the LUT. The **Truth Table** tab (see Figure 12–10) shows the truth table representations, and the **Karnaugh Map** tab (see Figure 12–11) shows the Karnaugh map representations of the LUT. The Karnaugh map supports up to 6 input LUTs. For details on the **Ports** tab, see "Viewing the Properties of Instances and Primitives".

Figure 12–9. Schematic Tab

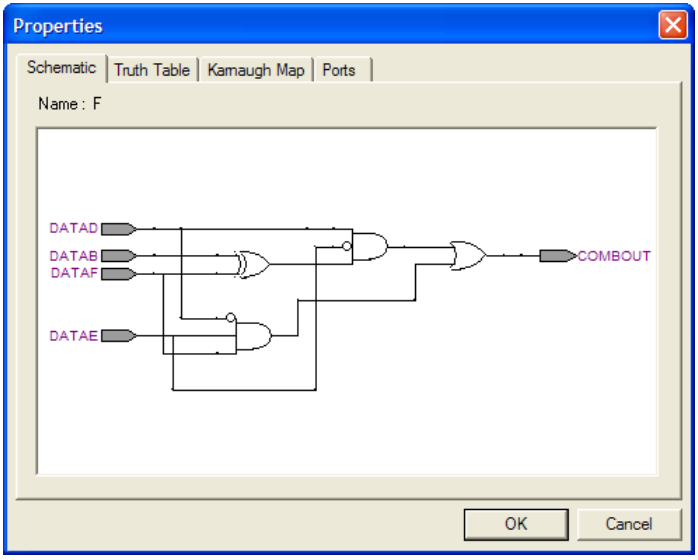
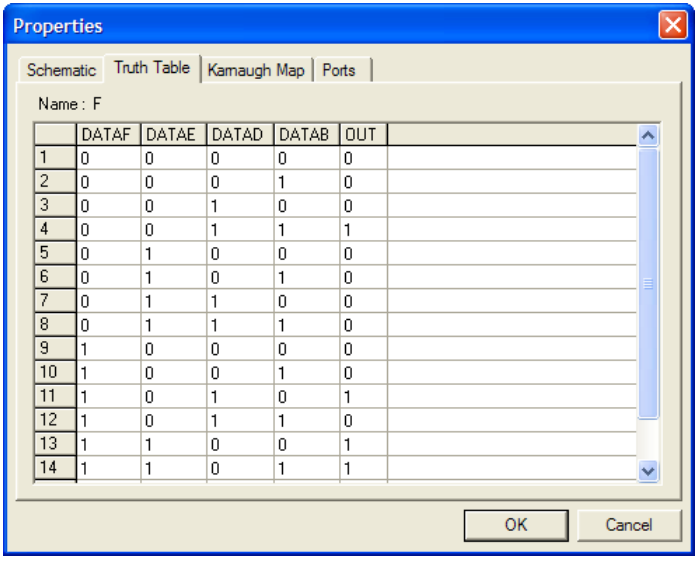
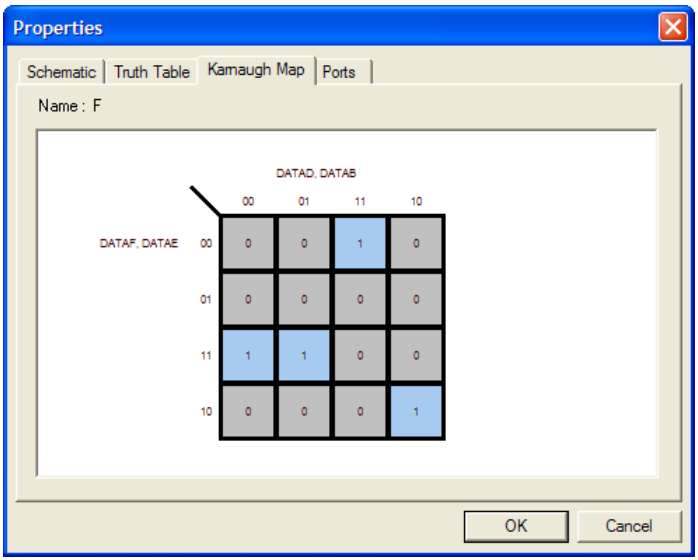


Figure 12–10. Truth Table Tab



	DATAF	DATAE	DATAD	DATAB	OUT
1	0	0	0	0	0
2	0	0	0	1	0
3	0	0	1	0	0
4	0	0	1	1	1
5	0	1	0	0	0
6	0	1	0	1	0
7	0	1	1	0	0
8	0	1	1	1	0
9	1	0	0	0	0
10	1	0	0	1	0
11	1	0	1	0	1
12	1	0	1	1	0
13	1	1	0	0	1
14	1	1	0	1	1

Figure 12–11. Karnaugh Map Tab



Zooming and Magnification

You can control the magnification of your schematic with the View menu, the **Zoom Tool** in the toolbar, or the Ctrl key and mouse wheel button, as described in this section.

The **Fit in Window**, **Fit Selection in Window**, **Zoom In**, **Zoom Out**, and **Zoom** commands are available from the View menu, by right-clicking in the schematic view and selecting **Zoom,** or from the **Zoom** toolbar. To enable the zoom toolbar, on the Tools menu, click **Customize**. Click the **Toolbars** tab and click **Zoom** to enable the toolbar.

By default, the viewer displays most pages sized to fit in the window. If the schematic page is very large, the schematic is displayed at the minimum zoom level, and the view is centered on the first node. Select **Zoom In** to view the image at a larger size, and select **Zoom Out** to view the image (when the entire image is not displayed) at a smaller size. The **Zoom** command allows you to specify a magnification percentage (100% is considered the normal size for the schematic symbols).

The **Fit Selection in Window** command zooms in on the selected nodes in a schematic to fit within the window. Use the **Selection Tool** to select one or more nodes (instances, primitives, pins, and nets), then select **Fit Selection in Window** to enlarge the area covered by the selection. This

feature is helpful when you want to see a particular element in a large schematic. After you select a node, you can easily zoom in to view the particular node.

You can also use the **Zoom Tool** on the viewer toolbar to control magnification in the schematic view. When you select the **Zoom Tool** in the toolbar, clicking on the schematic zooms in and centers the view on the location you clicked. Right-click on the schematic (or press the Shift key or the Ctrl key and click) to zoom out and center the view on the location you clicked. When you select the **Zoom Tool**, you can also zoom in to a certain portion of the schematic by selecting a rectangular box area with your mouse cursor. The schematic is enlarged to show the selected area. To change the minimum and the maximum zoom level, on the Tools menu, click **Options**. In the **Options** dialog box, in the **Category** list, select **Netlist Viewers**, and set the desired minimum and maximum zoom level.

By default, the viewers maintain the zoom level when filtering on the schematic (refer to "Filtering in the Schematic View" on page 12–34). To change the behavior so that the zoom level is always reset to "Fit in Window," on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers**, and turn off **Maintain zoom level**.

Schematic Debugging and Tracing Using the Bird's Eye View

Viewing the entire schematic can be useful when debugging and tracing through a large netlist. The Quartus II software allows you to view the entire schematic in a single window. The bird's eye view is displayed in a separate window that is linked directly to the netlist viewers. This feature is available in the RTL, Technology Map, and Technology Map (Post-Mapping) viewers.

The bird's eye view shows the current area of interest. Select the desired area by clicking and dragging the indicator or using the right-mouse button to form a rectangular box around the desired area. You can also click and drag the rectangular box to move around the schematic. To open the bird's eye view, on the View menu, click **Bird's Eye View**, or click on the Bird's Eye View icon in the Viewer toolbar (Figure 12–12).

Figure 12–12. Bird's Eye View Icon



Full Screen View

To set the viewer window to fill the whole screen, on the View menu, click **Full Screen**, or click the **Full Screen** icon in the viewer toolbar, or press Ctrl+Alt+Space. The keyboard shortcut toggles between the full screen and standard screen views.

Partitioning the Schematic into Pages

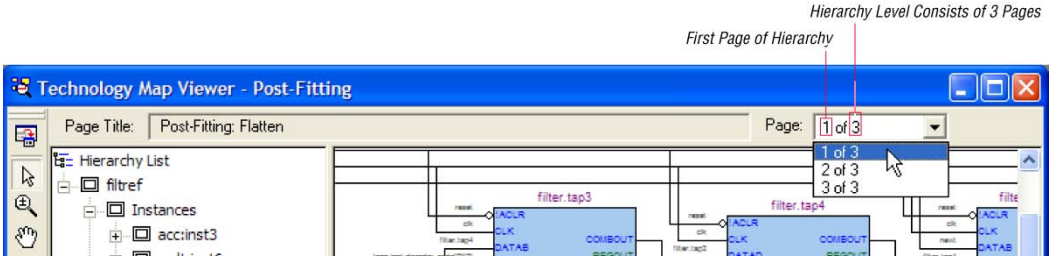
For large design hierarchies, the RTL Viewer and Technology Map Viewer partition your netlist into multiple pages in the schematic view. To control how much of the design is visible on each page, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers**, and set the desired options under **Display Settings**.

The **Nodes per page** option specifies the number of nodes per partitioned page. The default value is 50 nodes; the range is 1 to 1,000 nodes. The **Ports per page** option specifies the number of ports (or pins) per partitioned page. The default value is 1,000 ports (or pins); the range is 1 to 2,000 ports (or pins). The viewers partition your design into a new page if either the node number or the port number exceeds the limit you have specified. You may occasionally see the number of ports exceed the limit, depending on the configuration of nodes on the page.

If the **Display boundary around hierarchy levels** option is turned on, and the total number of nodes or ports within the hierarchy exceeds the value of **Nodes per page** or **Ports per page**, the boundary is displayed as a hierarchy port connector (refer to Table 12–1 on page 12–10). For more information about the **Display boundary around hierarchy levels** option, refer to "Filtering Across Hierarchies" on page 12–38.

When a hierarchy level is partitioned into multiple pages, the title bar for the schematic window indicates which page is displayed and how many total pages exist for this level of hierarchy (shown in the format: Page <*current page number*> of <*total number of pages*>), as shown in Figure 12–13.

Figure 12–13. RTL Viewer Title Bars Indicating Page Number Information



When you change the number of nodes or ports per page, the change applies only to new pages that are shown or opened in the viewer. To refresh the current page so that it displays the changed number of nodes or ports, click the **Refresh** button in the toolbar.

Moving Between Schematic Pages

To move to another schematic page, on the View menu, click **Previous Page** or **Next Page**, or click the **Previous Page** icon or the **Next Page** icon in the viewer toolbar.

To go to a particular page of the schematic, on the Edit menu, click **Go To**, or right-click in the schematic view, and click **Go To**. In the **Page** list, select the desired page number. You can also go to a particular page by selecting the desired page number from the drop down list on the top right of the viewer window.

Moving Back and Forward Through Schematic Pages

To return to the previous view after changing the page view, click **Back** on the View menu, or click the **Back** icon on the viewer toolbar. To go to the next view, click **Forward** on the View menu, or click the **Forward** icon on the viewer toolbar.



You can go **Forward** only if you have not made any changes to the view since going **Back**. Use **Back** and **Forward** to switch between page views. These commands do not undo an action such as selecting a node.

Following Nets Across Schematic Pages

Input and output connectors indicate nodes that connect across pages of the same hierarchy. Right-click on a connector to display a menu of commands that trace the net through the pages of the hierarchy.

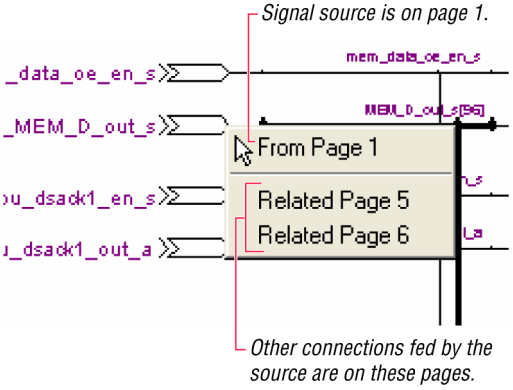


After you right-click to follow a connector port, the viewer opens a new page, which centers the view on the particular source or destination net using the same zoom factor used by the previous page. To trace a specific net to the new page of the hierarchy, Altera recommends that you first select the desired net, which highlights it in red, before you right-click to traverse pages.

Input Connectors

Figure 12–14 shows an example of the menu that appears when you right-click an input connector. The **From** command opens the page containing the source of the signal. The **Related** commands, if applicable, open the specified page containing another connection fed by the same source.

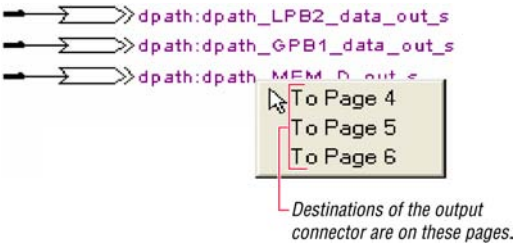
Figure 12–14. Input Connector Right Button Pop-Up Menu



Output Connectors

Figure 12–15 shows an example of the menu that appears when you right-click an output connector. The **To** command opens the specified page that contains a destination of the signal.

Figure 12–15. Output Connector Right Button Pop-Up Menu



Go to Net Driver

To locate the source of a particular net in the schematic view, select the net to highlight it, right-click the selected net, point to Go to Net Driver, and click **Current page**, **Current hierarchy**, or **Across hierarchies**. Refer to Table 12–6 for details.

Table 12–6. Go to Net Driver Commands

Command	Action
Current page	Locates the source or driver on the current page of the schematic only.
Current hierarchy	Locates the source within the current level of hierarchy, even if the source is located on another page of the netlist schematic.
Across hierarchies	Locates the source across hierarchies until the software reaches the source at the top hierarchy level.

The schematic view opens the correct page of the schematic if needed, and adjusts the centering of the page so that you can see the net source. The schematic shows the default page for the net driver. The view is an unfiltered view, so no filtering results are kept.

Customizing the Schematic Display in the RTL Viewer

You can customize the schematic display for better viewing and to speed up your debugging process. The options that control the schematic display are available in the **Customize View** tab of the **RTL/Technology Map Viewer Options** dialog box. To open the dialog box, right-click in the schematic and click **Viewer Options**. You can turn on the options to remove fan-out free nodes, simplify logic, group or ungroup related nodes, and group combinational logic into a logic cloud.

You can also customize the schematic view in the RTL Viewer by clicking **Options** on the Tools menu, in the Category list expand Netlist Viewers, and selecting **RTL Viewer**. Set the desired customization for your schematic display.



When the settings are changed, the list of previously viewed pages is cleared. The settings are revision-specific, so different revisions could have different settings.

To remove fan-out free registers from your schematic display, turn on **Remove registers without fan-out**. By default, this option is turned on.

To remove all single-input nodes and merge a chain of equivalent combinational gates that have direct connections (without inversion in between) into a single multiple-input gate, turn on **Show simplified logic.** By default, this option is turned on.

To group all related nodes into a single node, turn on **Group all related nodes**. This option is turned on by default. You can manually group or ungroup any nodes by right-clicking the selected nodes in the schematic and selecting **Group Related Nodes** to group, or **Ungroup Selected Nodes** to ungroup.

Grouping Combinational Logic into Logic Clouds

You can automatically group all combinational logic nodes in your design into logic clouds by clicking **Options** on the Tools menu, in the Category list expand Netlist Viewers, and select **RTL Viewer**. In the **RTL Viewer** page, turn on **Group combinational logic in logic cloud**. You can also set this option by right-clicking in the schematic and click **Viewer Options**. In the **RTL/Technology Map Viewer Options** dialog box, click on the **Customize View** tab. In the Customize Groups section, turn on the **Group combinational logic in logic cloud** option. Figure 12–16 and Figure 12–17 show the schematic before and after the combinational logic grouping operation.

Figure 12–16. Schematic Before Combinational Logic Grouping

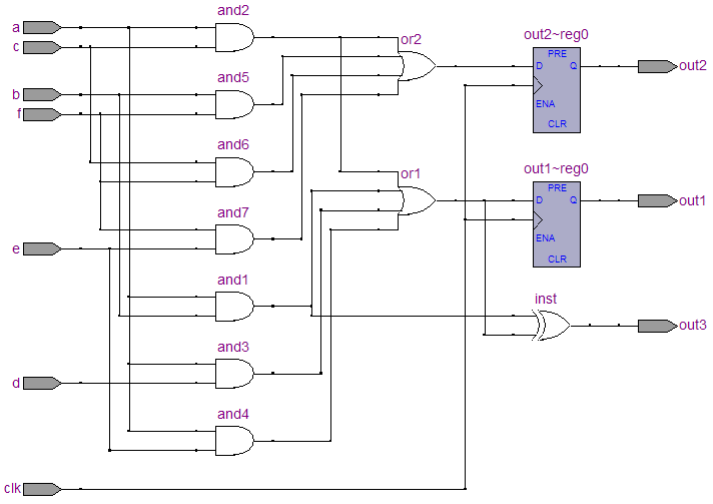
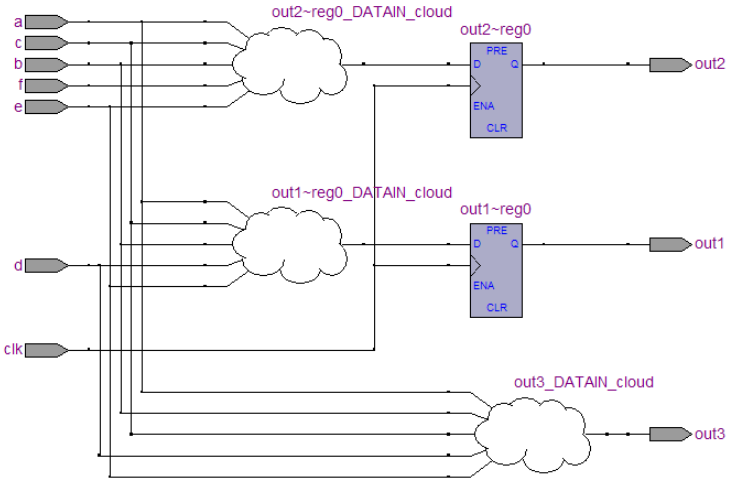


Figure 12–17. Schematic After Combinational Logic Grouping



To manually group combinational logic nodes into a logic cloud, right-click the selected node or input port, and select **Group Source Logic into Logic Cloud**. To manually ungroup logic cloud, right-click on the selected logic cloud and select **Ungroup Source Logic from Logic Cloud**. You can also manually ungroup a logic cloud by double-clicking on the selected logic cloud. These options are not available if the nodes cannot be grouped.

Filtering in the Schematic View

Filtering allows you to filter out nodes and nets in your netlist to view only the logic that interests you.

Filter your netlist by selecting hierarchy boxes, nodes, ports of a node, net, or states in a state machine that are part of the path you want to see. The following filter commands are available:

- **Sources**—Displays the sources of the selection
- **Destinations**—Displays the destinations of the selection
- **Sources & Destinations**—Displays both the sources and destinations of the selection
- **Selected Nodes and Nets**—Displays only the selected nodes and nets with the connections between them
- **Between Selected Nodes**—Displays nodes and connections in the path between the selected nodes
- **Bus Index**—Displays the sources or destinations for one or more indices of an output or input bus port

Select a hierarchy box, node, port, net, or state node, right-click in the window, point to Filter and click the appropriate filter command. The viewer generates a new page showing the netlist that remains after filtering.

When filtering in a state diagram in the State Machine Viewer, sources and destinations refer to the previous and next transition states or paths between transition states in the state diagram. The transition table and encoding table also reflect the filtering.

You can go back to the netlist page before it was filtered using the **Back** command, described in "Moving Back and Forward Through Schematic Pages" on page 12–29.



When viewing a filtered netlist, clicking an item in the hierarchy list causes the schematic view to display an unfiltered view of the appropriate hierarchy level. You cannot use the hierarchy list to select items or navigate in a filtered netlist.

Filter Sources Command

To filter out all but the source of the selected item, right click the item, point to Filter and click **Sources**. The selected object type determines what is displayed, as outlined in Table 12–7, and shown in Figure 12–18 on page 12–36.

Table 12–7. Selected Objects Determine Filter Sources Display

Selected Object	Result Shown in Filtered Page
Node or hierarchy box	Shows all the sources of the node's input ports. For an example, refer to Figure 12–18 on page 12–36.
Net	Shows the sources that feed the net.
Input port of a node	Shows only the input source nodes that feed this port.
Output port of a node	Shows only the selected node.
State node in a state machine	Shows the states that feed the selected state (previous transition states).

Filter Destinations Command

To filter out all but the destinations of the selected node or port as outlined in Table 12–8, and shown in Figure 12–18 on page 12–36, right-click the node or port, point to Filter, and click **Destinations**.

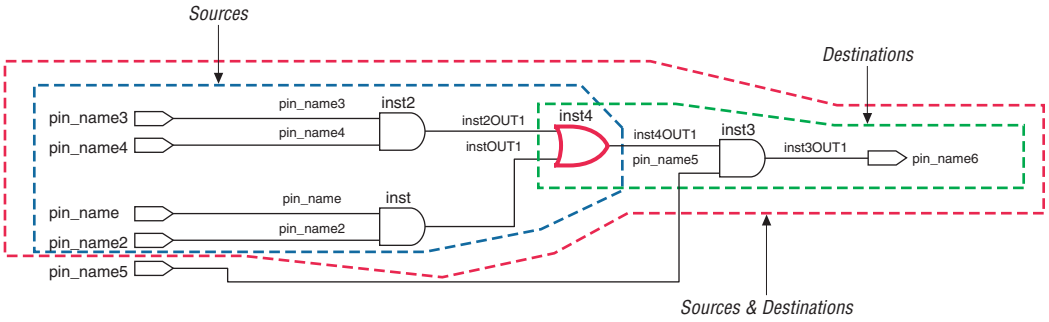
Table 12–8. Selected Objects Determine Filter Destinations Display

Selected Object	Result Shown in Filtered Page
Node or hierarchy box	Shows all the destinations of the node's output ports. For an example, refer to Figure 12–18 on page 12–36.
Net	Shows the destinations fed by the net.
Input port of a node	Shows only the selected node.
Output port of a node	Shows only the fan-out destination nodes fed by this port.
State node in a state machine	Shows the states that are fed by the selected states (next transition states).

Filter Sources and Destinations Command

The **Sources & Destinations** command is a combination of the **Sources** and **Destinations** filtering commands, in which the filtered page shows both the sources and the destinations of the selected item. To select this option, right-click on the desired object, point to Filter, and click **Sources & Destinations**. Refer to the example in Figure 12–18.

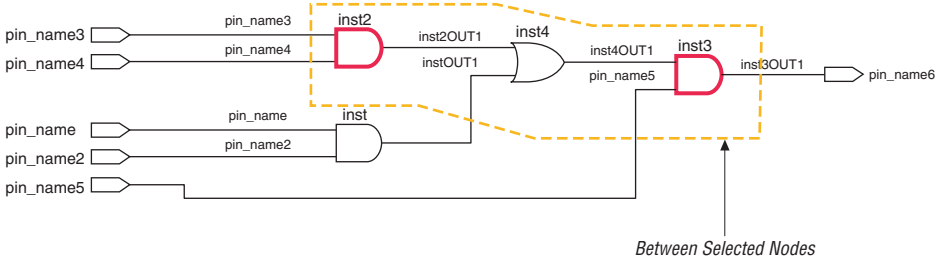
Figure 12–18. Sources, Destinations, and Sources and Destinations Filtering for inst4



Filter Between Selected Nodes Command

To show the nodes in the path between two or more selected nodes or hierarchy boxes, right-click, point to Filter, and click **Between Selected Nodes**. For this option, selecting a port of a node is the same as selecting the node. For an example, refer to Figure 12–19.

Figure 12–19. Between Selected Nodes Filtering Between inst2 and inst3



Filter Selected Nodes and Nets Command

To create a filtered page that shows only the selected nodes, nets, or both, and, if applicable, the connections between the selected nodes, nets, or both, right-click, point to Filter, and click **Selected Nodes & Nets**. Figure 12–20 shows a schematic with several nodes selected.

Figure 12–20. Using Selected Nodes and Nets to Select Nodes

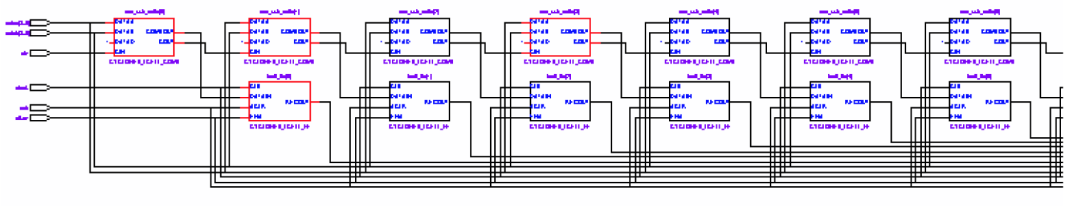
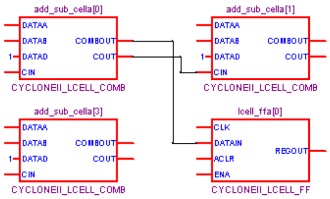


Figure 12–21 shows the schematic after filtering has been performed. If you select a net, the filtered page shows the immediate sources and destinations of the selected net.

Figure 12–21. Selected Nodes and Nets Filtering on Figure 12–20 Schematic



Filter Bus Index Command

To show the path related to a specific index of a bus input or output port in the RTL Viewer, right-click the port, point to Filter, and click **Bus Index**. The **Select Bus Index** dialog box allows you to select the indices of interest.

Filter Command Processing

The options to control filtering are available in the **Tracing** section of the **RTL/Technology Map Viewer Options** dialog box. Right-click in the schematic, and click **Viewer Options** to open the dialog box.

For all the filtering commands, the viewer stops tracing through the netlist to obtain the filtered netlist when it reaches one of the following objects:

- A pin
- A specified number of filtering levels, counting from the selected node or port; the default value is 3

  Specify the **Number of filtering** levels in the **Tracing** section of the **RTL/Technology Map Viewer Options** dialog box. The default value is 3 to ensure optimal processing time when performing filtering, but you can specify a value from 1 to 100.

- A register (optional; turned on by default)

  Turn the **Stop filtering at register** option on or off in the **Tracing** section of the **RTL/Technology Map Viewer Options** dialog box. Right-click in the schematic and click **Viewer Options** to open the dialog box.

By default, the filtered schematic shows all possible connections between the nodes shown in the schematic. To remove the connections that are not directly part of the path that was traced to generate a filtered netlist, turn off the **Shows all connections between nodes** option in the Tracing section of the **RTL/Technology Map Viewer Option**s dialog box.

Filtering Across Hierarchies

The filtering commands display nodes in all hierarchies by default. When the filtered path passes through levels of hierarchy on the same schematic page, green hierarchy boxes group the logic and show the hierarchy boundaries. A green rectangular symbol appears on the border that represents the port relationship between two different hierarchies (Figure 12–22 and Figure 12–23).

The **RTL/Technology Map Viewer Options** dialog box provides an option to control filtering if you prefer to filter only within the current hierarchy. Right-click in the schematic, and click **Viewer Options**. In the **Tracing** section, turn off the **Filter across hierarchy** option.

To disable the box hierarchy display, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and turn off **Display boundary around hierarchy levels**.



Netlists of the same hierarchy that are displayed over more than one page are not grouped with a box. Filtering and expanding on a blue atom primitive does not trace the underlying netlist even when **Filter across hierarchy** is enabled.

Figures 12–22 and 12–23 show examples of filtering across hierarchical boundaries. Figure 12–22 shows an example after the **Sources** filter has been applied to an input port of the `taps` instance, where the input port of the lower level hierarchical block connects directly to an input pin of the design. The name of the instance is indicated within the green border and appears as a tooltip when you move your mouse pointer over the instance.

Figure 12–22. Filtering Across Hierarchical Boundaries, Small Example

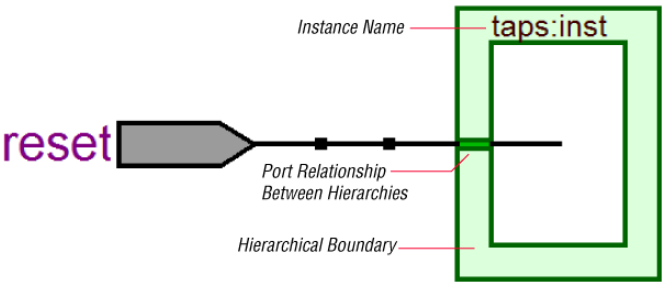
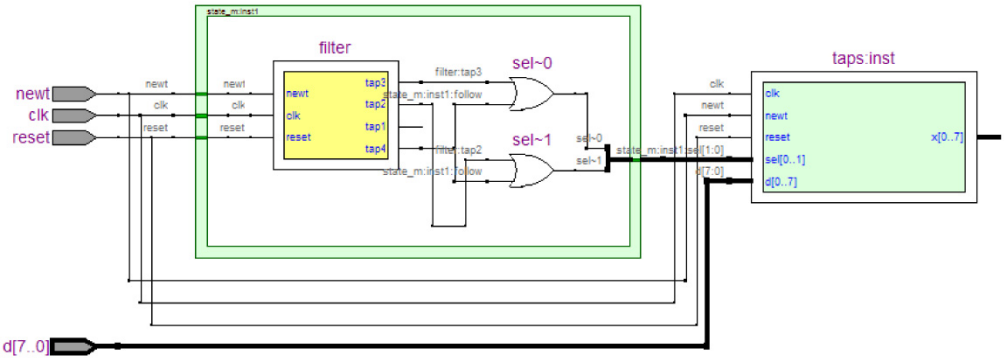


Figure 12–23 shows a larger example after the **Sources** filter has been applied to an input port of an instance, in which the source comes from input pins that are fed through another level of hierarchy.

Figure 12–23. Filtering Across Hierarchical Boundaries, Large Example



Sources command applied to an input port of an instance in which the source comes from input pins that are fed through another level of hierarchy.

Expanding a Filtered Netlist

After a netlist is filtered, some ports may have no connections displayed because their connections are not part of the main path through the netlist. Two expansion features, immediate expansion and the **Expand** command, allow you to add the fan-in or fan-out signals of these ports to the schematic display of a filtered netlist.

You can immediately expand any port whose connections are not displayed. When you double-click that port in the filtered schematic, one level of logic is expanded.

To expand more than one level of logic, right-click the port and click the **Expand** command. This command expands logic from the selected port by the amount specified in the **Viewer Options**. To set these options, right-click in the schematic view, and click **Viewer Options**. In the Expansion section, set the **Number of expansion levels** option to specify the number of levels to expand (the default value is 3 and the range is 1 to 100 levels). You can also set the **Stop expanding at register** option (which is turned on by default) to specify whether netlist expansion should stop when a register is reached.

You can select multiple nodes to expand when using the **Expand** command. If you select ports that are located on multiple schematic pages, only the ports on the currently viewed page appear in the expanded schematic.

In the State Machine Viewer, the **Expand** command has the following three options:

- **Sources**—Displays the states that feed the selected states (previous transition states)
- **Destinations**—Displays the states that are fed by the selected states (next transition states)
- **Sources & Destinations**—Displays both the previous and next transition states

The state transition table and state encoding table also reflect the changes to the filtering.

The expansion feature works across hierarchical boundaries if the filtered page containing the port to be expanded was generated with the **Filter across hierarchy** option turned on (refer to "Filtering in the Schematic View" on page 12–34 for details on this option). When viewing timing paths in the Technology Map Viewer, the **Expand** command always works across hierarchical boundaries because filtering across hierarchy is always turned on for these schematics (refer to "Viewing a Timing Path" on page 12–45 for details on these schematics).

Reducing a Filtered Netlist

In some cases, removing logic from a filtered schematic or state diagram makes the schematic view easier to read or minimizes distracting logic that you do not need to view in the schematic.

To reduce elements in the filtered schematic or state diagram view, right-click the node or nodes you want to remove and click **Reduce**.

Probing to Source Design File and Other Quartus II Windows

The RTL, Technology Map, and State Machine Viewers let you cross-probe from the viewer to the source design file and to various other windows within the Quartus II software. You can select one or more hierarchy boxes, nodes, nets, state nodes, or state transition arcs that interest you in the viewer and locate the corresponding items in another applicable Quartus II software window. You then can view and make changes or assignments in the appropriate editor or floorplan.

To locate an item from the viewer in another window, right-click the items of interest in the schematic or state diagram view, point to Locate, and click the appropriate command. The following commands are available:

- **Locate in Assignment Editor**
- **Locate in Pin Planner**
- **Locate in Timing Closure Floorplan**
- **Locate in Chip Planner**
- **Locate in Resource Property Editor**
- **Locate in RTL Viewer**
- **Locate in Technology Map Viewer**
- **Locate in Design File**

The options available for locating depend on the type of node and whether it exists after placement and routing. If a command is enabled in the menu, then it is available for the selected node. You can use the **Locate in Assignment Editor** command for all nodes, but assignments may be ignored during placement and routing if they are applied to nodes that do not exist after synthesis.

The viewer automatically opens another window for the appropriate editor or floorplan, and highlights the selected node or net in the newly opened window. You can switch back to the viewer by selecting it in the Window menu or by closing, minimizing, or moving the new window.



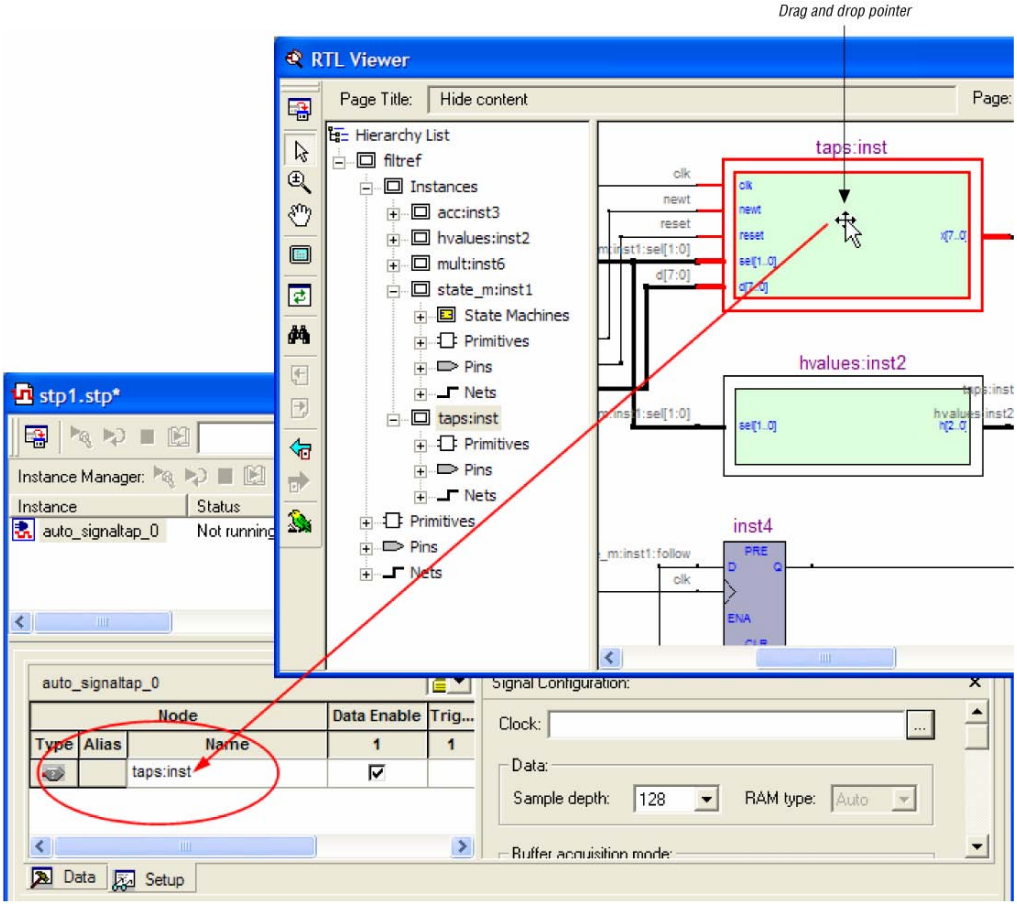
When probing to a logic cloud in the RTL Viewer, a message box appears that prompts you to ungroup the logic cloud or allow it to remain grouped.

Moving Selected Nodes to Other Quartus II Windows

You can drag selected nodes from the netlist viewers to the Text Editor, Block Editor, Pin Planner, SignalTap® II, and Waveform Editor windows within the Quartus II software. Whenever you see the drag-and-drop pointer on the selected node in the netlist viewers, it means that the node can be dragged to other child windows within the Quartus II software.

Figure 12–24 shows the drag-and-drop pointer and an example of dragging a node from the RTL Viewer to the SignalTap II Logic Analyzer.

Figure 12–24. Dragging a Node to the SignalTap II Logic Analyzer



Probing to the Viewers from Other Quartus II Windows

You can cross-probe to the RTL Viewer and Technology Map Viewer from other windows within the Quartus II software. You can select one or more nodes or nets in another window and locate them in one of the viewers.

You can locate nodes between the RTL, State Machine, and Technology Map Viewers, and you can locate nodes in the RTL Viewer or Technology Map Viewer from the following Quartus II software windows:

- Project Navigator
- Timing Closure Floorplan
- Chip Planner
- Resource Property Editor
- Node Finder
- Assignment Editor
- Messages Window
- Compilation Report
- TimeQuest Timing Analyzer (only supports the Technology Map Viewer)

To locate elements in the viewer from another Quartus II window, select the node or nodes in the appropriate window; for example, select an entity in the **Entity** list on the **Hierarchy** tab in the Project Navigator, or select nodes in the **Timing Closure Floorplan**, or select node names in the **From** or **To** column in the Assignment Editor. Next, right-click the selected object, point to Locate, and click **Locate in RTL Viewer** or **Locate in Technology Map Viewer**. After you choose this command, the viewer window opens, or is brought to the foreground if the viewer window is already open.



The first time the window opens after a compilation, the preprocessor stage runs before the viewer window opens.

The viewer shows the selected nodes and, if applicable, the connections between the nodes. The display is similar to what you see if you right-click the object, point to Filter, and click **Selected Nodes & Nets** using **Filter Across Hierarchy**. If the nodes cannot be found in the viewer, a message box displays the message: “Can’t find requested location.”

Viewing a Timing Path

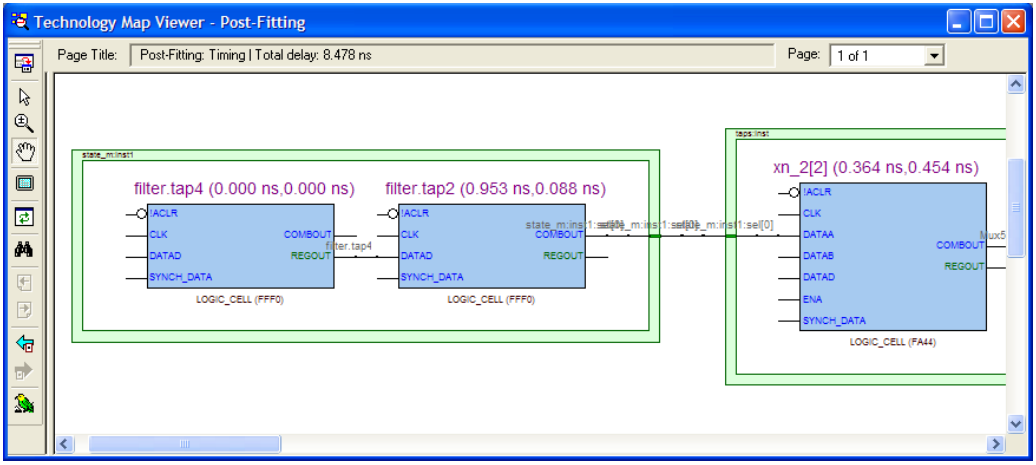
To see a visual representation of a timing path, you can cross-probe from the Timing Analysis section of the Compilation Report with the Classic Timing Analyzer, or from a report panel in the TimeQuest Timing Analyzer.

To take advantage of this feature, you must first successfully complete a full compilation of your design, including the timing analyzer stage. To access the timing analyzer report that contains the timing results for your design, on the Processing menu, click **Compilation Report**. On the left side of the Compilation Report, select **Timing Analyzer** or **TimeQuest Timing Analyzer**. When you select a detailed report, the timing information is listed in a table format on the right side of the Compilation Report; each row of the table represents a timing path in the design. You can also view timing paths in TimeQuest report panels. To view a particular timing path in the Technology Map Viewer or the RTL Viewer, highlight the appropriate row in the table, right-click, point to Locate, and click **Locate in Technology Map Viewer** or **Locate in RTL Viewer**.

In the Technology Map Viewer, the schematic page displays the nodes along the timing path with a summary of the total delay. If you locate from the Classic Timing Analyzer, the timing path also includes timing data representing the interconnect (IC) and cell delays associated with each node. The delay for each node is shown in the following format: <*post-synthesis node name*> (<*IC delay*> ns, <*cell delay*> ns). When you locate the timing path from the TimeQuest Timing Analyzer to the Technology Map Viewer, the interconnect and cell delay associated with each node is not displayed.

Figure 12–25 shows a portion of a Classic Timing Analyzer timing path represented in the Technology Map Viewer. The total delay for the entire path through several levels of logic (only three levels are shown in Figure 12–25) is 7.159 ns. The delays are indicated for each level of logic. For example, the IC delay to the first `LCELL` primitive is 0.383 ns and the cell delay through the `LCELL` is 0.075 ns. When the timing path passes through a level of hierarchy, green hierarchy boxes group the logic and show the hierarchical boundaries. A green rectangular symbol on the border indicates the path passes between two different hierarchies.

Figure 12–25. Timing Path Schematic in the Technology Map Viewer



In the RTL Viewer, the schematic page displays the nodes in the path(s) between the source and destination registers with a summary of the total delay.

The RTL Viewer netlist is based on an initial stage of synthesis, so the post-fitting nodes may not exist in the RTL Viewer netlist. Therefore, the internal delay numbers are not displayed in the RTL Viewer as they are in the Technology Map Viewer, and the timing path may not be displayed exactly as it appears in the timing analysis report. If multiple paths exist between the source and destination registers, the RTL Viewer may display more than just the timing path. There are also some cases in which the path cannot be displayed, such as paths through state machines, encrypted intellectual property (IP), or registers that are created during the fitter process. In cases where the timing path displayed in the RTL Viewer might not be the correct path, the compiler issues messages.

Other Features in the Schematic Viewer

This section describes other features in the schematic view that enhance usability and help you analyze your design.

Tooltips

A tooltip is displayed whenever the mouse pointer is held over an element in the schematic. The tooltip contains useful information about a node, net, logic cloud, input port, and output port. Table 12–9 lists the information contained in the tooltip for each type of node.

The tooltip information for an instance (the first row in Table 12–9) includes a list of the primitives found within that level of hierarchy, and the number of each primitive contained in the current instance. The number includes all hierarchical blocks below the current instance in the hierarchy. This information lets you estimate the size and complexity of a hierarchical block without navigating into the block.

The tooltip information for atom primitives in the Technology Map Viewer (the second row of Table 12–9) shows the equation for the design atom. The equations are an expanded version of the equations you can view in the Equations window in the Timing Closure Floorplan. Advanced users can use these equations to analyze the design implementation in detail.



For details on understanding equations, refer to the Quartus II Help.

To copy tooltips into the clipboard for use in other applications, right-click the desired node or netlist, and click **Copy Tooltip**.

To turn off tooltips or change the duration of time that a tooltip is displayed in the view, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and set the desired options under **Tooltip settings**.

The **Show names in tooltip for** option specifies the number of seconds to display the names of assigned nodes and pins in a tooltip when the pointer is over the assigned nodes and pins. Selecting **Unlimited** displays the tooltip as long as the pointer remains over the node or pin. Selecting **0** turns off tooltips. The default value is 5 seconds.

The **Delay showing tooltip for** option specifies the number of seconds you must hold the mouse pointer over assigned nodes and pins before the tooltip displays the names of the assigned nodes and pins. Selecting **0** displays the tooltip immediately when the pointer is over an assigned node or pin. Selecting **Unlimited** prevents tooltips from being displayed. The default value is 1 second.

Table 12–9. Tooltip Information (Part 1 of 2)

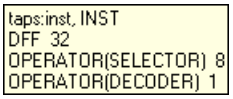
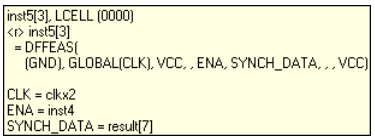
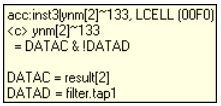
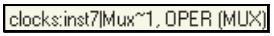
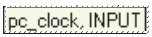
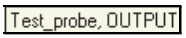
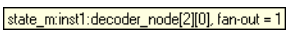
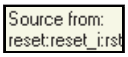
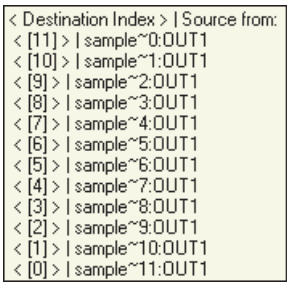
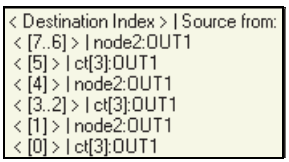
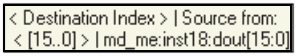
Description and Tooltip Format	Example Tooltips
Instance Format: <*instance name*>, <*instance type*> <*primitive type*>, <*number of primitives*>... <*primitive type*>, <*number of primitives*>	
Atom Primitive Format: <*instance name*> , <*primitive name*> (<*LUT Mask Value*>) {(`r` | `c` <*Register or Combinational equation*>)} ... An `r` (as in the first example) represents the equation for a register, and a `c` (as in the second example) represents the equation for combinational logic.	 
Primitive Format:<*primitive name*>, <*primitive type*>	 
Pin Format: <*pin name*>, <*pin type*>	 
Connector Format: <*connector name*>	
Net Format: <*net name*>, `fan-out` = <*number of fan-out signals*>	
Output Port Format: `fan-out` = <*number of fan-out signals*>	

Table 12–9. Tooltip Information (Part 2 of 2)

Description and Tooltip Format	Example Tooltips
Input Port The information displayed depends on the type of source net. The examples of the tooltips shown represent the following types of source nets: *(1)* Single net *(2)* Individual nets, part of the same bus net *(3)* Combination of different bus nets *(4)* Constant inputs *(5)* Combination of single net and constant input *(6)* Bus net Source from—refers to the source net name that connects to the input port. Destination Index—refers to the bit(s) at the destination input port to which the source net is connected (not applicable for single nets).	 *(1)*  *(2)*  *(3)*  *(4)*  *(5)*  *(6)*
State Machine Node Format: <*node name*>	
State Machine Transition Arc This information is displayed when you hold your mouse over the arrow on the arc representing the transition between two states. Format: (<*equation for transition between states*>)	

Radial Menu

This user interface feature provides a quick method to perform some shortcut commands in the schematic view. The commands are listed in an octagon-shaped menu, and you can perform the commands via mouse stroke.

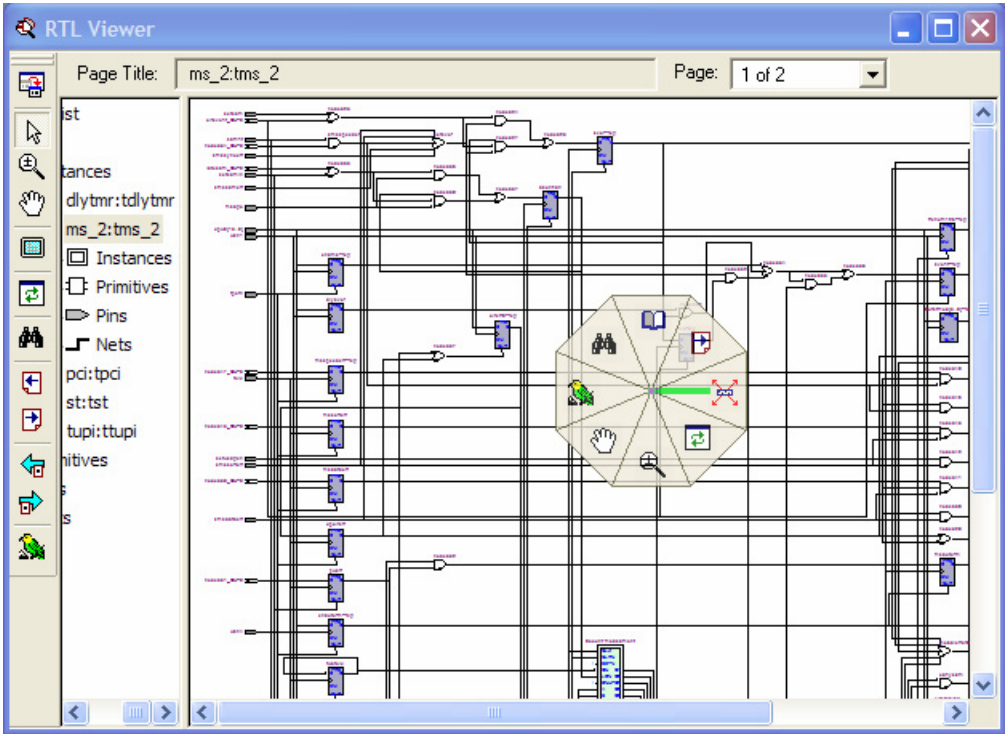
To launch the Radial menu, hold down the CTRL key and right-click the mouse anywhere in the schematic view. The Radial menu appears with the mouse cursor always at the center point. To trigger the desired command, you can do either of the following:

- While holding the CTRL key, click the desired region/command in the Radial menu to execute the command.
- While holding the CTRl key, right-click and hold the mouse button, drag the pointer onto the desired command, and then release to execute the command.

However, if you release the CTRL key while performing either of the above actions, the Radial menu disappears, without executing the command.

Figure 12–26 shows the radial menu in action.

Figure 12–26. Radial Menu

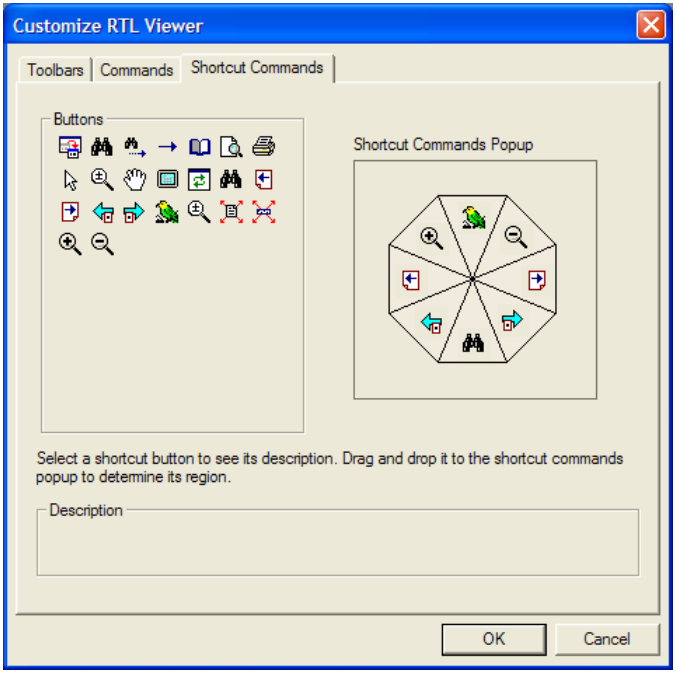


Customizing the Radial Menu

You can customize all eight commands in the Radial menu. To customize the radial menu, you first need to launch the RTL Viewer or the Technology Map Viewer. Then on the Tools menu, click **Customize RTL Viewer**, **Customize Technology Map Viewer**, or **Customize Technology Map Viewer (Post-Mapping)** and click on the Shortcut Commands tab. The Buttons section of the dialog box shows a list of Netlist Viewer commands that you can choose. You can click on a command to see its description in the Description section. To make the desired command appear on the radial menu, drag-and-drop the command onto the Radial menu diagram in the Shortcut Commands Popup section. Repeated commands are allowed in the radial menu.

Figure 12–27 shows the Shortcut Commands tab for customizing the radial menu.

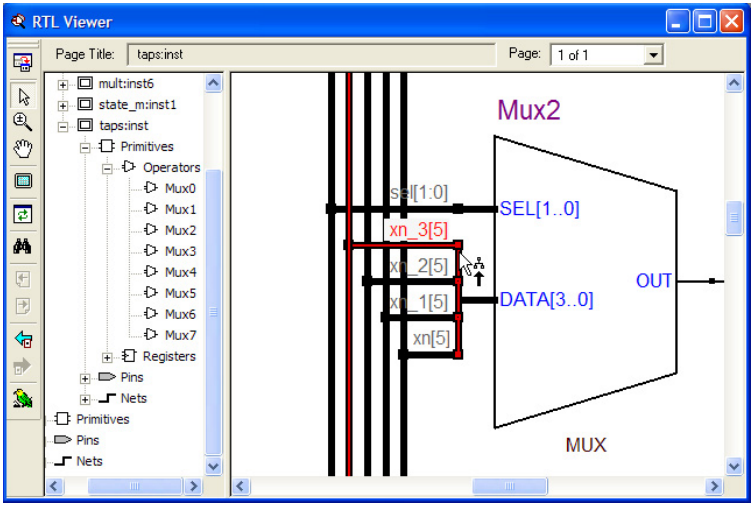
Figure 12–27. Shortcut Command Tab



Rollover

You can highlight an element and view its name in your schematic using the rollover feature. When you place your mouse pointer over an object, the object is highlighted and the name is displayed (Figure 12–28). This feature is enabled by default in the netlist viewers. To turn off the Rollover feature, on the Tools menu, click **Options**. In the **Options** dialog box, in the **Category** list, select **Netlist Viewers** and turn off **Enable Rollover**.

Figure 12–28. Rollover in the RTL Viewer and Technology Map Viewer



Displaying Net Names

To see the names of all the nets displayed in your schematic, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and turn on **Show Net Name** under **Display Settings**. This option is disabled by default. If you turn on this option, the schematic view refreshes automatically to display the net names.

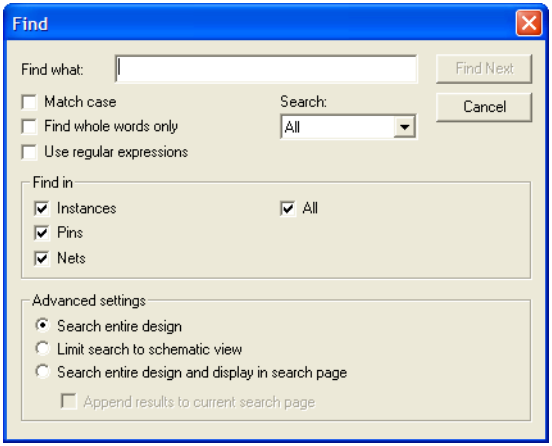
Displaying Node Names

In some designs, nodes have long names that overlap the ports of other symbols in the schematic. To remove the node names from the schematic, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and turn off **Show node name** under **Display Settings**. This option is turned on by default.

Find Command

To open the **Find** dialog box shown in Figure 12–29, on the Edit menu, click **Find**, or click the **Find** icon in the viewer toolbar, or right-click in the schematic view, and click **Find**.

Figure 12–29. Find Dialog Box



You can choose to search only instances (nodes) in the design, or to also search pins and nets. By default, only instances are searched.

When you click **Find**, the viewer selects and highlights the first item found, opens the appropriate page of the schematic, if necessary, and centers the page so that the node is visible in the viewable area (but does not zoom in to the node). To find the next matching node, click **Find Next**. When the node that you search for is part of a logic cloud, the logic cloud that contains the node is highlighted. A message box appears that prompts you to ungroup the logic cloud or allow it to remain grouped.

You can use the options in the Advanced settings section to control the scope of the results found during a search and how they are displayed in the viewer. The default selection, **Search entire design**, searches for the item in all design elements across the entire design. To search only in the pages of the currently displayed netlist, such as a schematic showing filtering results, choose **Limit search to schematic view**.

To display the results in a new page, select **Search entire design and display in search page**. This command searches all design elements across the entire design, and displays the results on a separate page dedicated to search results. You can also append new search results to an existing search page with the **Append results to current search page** command. The appended items appear in the same relative position as they do in the full schematic. You can use this method to find and select two objects that are not on the same page and display them on the same page after performing the **Find** command.



Refer to "Finding Nodes in the RTL Viewer and Technology Map Viewer" in the Quartus II Help for more details about using the **Find** dialog box.

Exporting and Copying a Schematic Image

You can export the RTL Viewer or Technology Map Viewer schematic view in JPEG File Interchange Format (**.jpg**) or Windows Bitmap (**.bmp**) file format, which allows you to include the schematic in project documentation or share it with other project members. To export the schematic view, on the File menu, click **Export**. In the **Export** dialog box, type a file name and location, and select the desired file type. The default file name is based on the current instance name and the default file type is JPEG Interchange Format (**.jpg**). However, for pages that use filtering, expanding, or reducing operations, the default name is **Filter**<*number of export operation*>.jpg. Nodes grouped as logic clouds are not shown in the exported or copied schematic image.

You can copy the whole image or only a portion of the image. To copy the full image, on the Edit menu, point to Copy and click **Full Image**. To copy a portion of the image, on the Edit menu, point to Copy and click **Partial Image**. The cursor changes to a plus sign to indicate that you can draw a box shape. Drag the cursor around the portion of the schematic you want to copy. When you release the mouse button, the partial image is copied to the clipboard.



Occasionally, due to the design size and objects selected, an image is too large to copy to the clipboard. In this case, the Quartus II software displays an error message.

To export or copy a schematic that is too large to copy in one piece, first split the design into multiple pages to export or to copy smaller portions of the design. For information about how to control how much of your design is shown on each schematic page, refer to "Partitioning the Schematic into Pages" on page 12–28. As an alterative, use the Partial Image feature to copy a portion of the image.

The **Copy** feature is not available on UNIX platforms.

Printing

To print your schematic page, on the File menu, click **Print**. You can print each schematic page onto one full page, or you can print the selected parts of your schematic onto one page with the **Selection** option. Refer to "Partitioning the Schematic into Pages" on page 12–28 to control how much of your design is shown on each schematic page.

Before printing, you can modify the page orientation. On the File menu, click **Page Setup**. Change the page orientation from **Portrait** to **Landscape**, or to the setting that best fits your design. You can also adjust the page margins in the **Page Setup** dialog box.

The hierarchy list in the viewers and the table view of the State Machine Viewer cannot be printed. You can use the State Machine Viewer **Copy** command to copy the table to a text editor and print from the text editor.

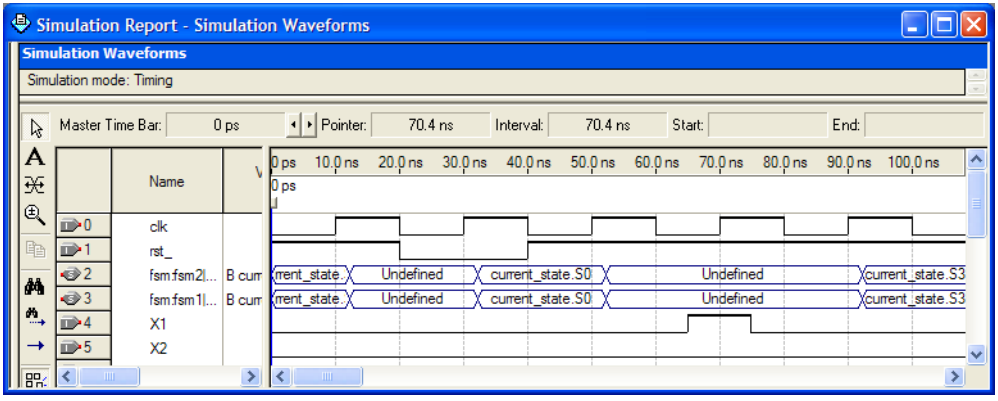
Debugging HDL Code with the State Machine Viewer

This section provides an example of using the State Machine Viewer to help debug HDL code. This example shows how you can use the various features in the netlist viewers to help solve design problems.

Simulation of State Machine Gives Unexpected Results

This section presents a design scenario in which you compiled your design and performed a simulation in the Quartus II Simulator. The simulation result is shown in Figure 12–30 and has unexpected undefined states.

Figure 12–30. Simulation Result Showing Undefined States



To analyze the state machine design in the State Machine Viewer, follow these steps:

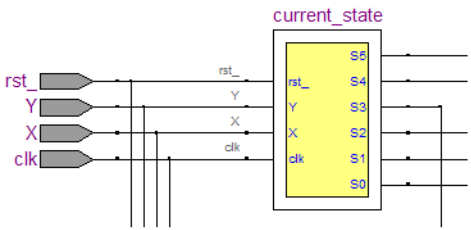
1. Open the State Machine Viewer for the state machine of interest. You can do this in any of the following ways:

- On the Tools menu, point to Netlist Viewers and click **State Machine Viewer**. In the State Machine selection box, choose the state machine that you want to view.

- On the Tools menu, point to Netlist Viewers, and click **RTL Viewer**. Browse to the hierarchy block that contains the state machine definition and double-click the yellow state machine instance to open the State Machine Viewer (Figure 12–31).You can open the State Machine Viewer using either of two methods:

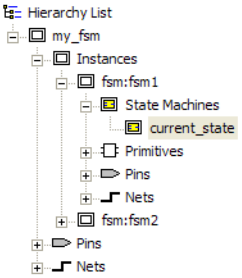
  - In the schematic view, double-click an instance in the hierarchy to open the lower level hierarchy. You can traverse through the schematic hierarchy in this way to open the schematic page that contains the state machine (Figure 12–31).

Figure 12–31. State Machine Instance in RTL Viewer Schematic View



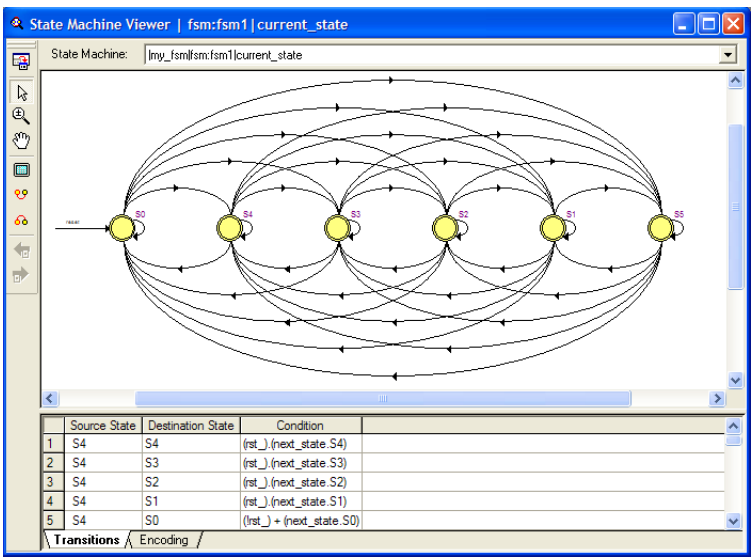
  - In the hierarchy list, click the **+** symbol next to **Instances** to open a list of the instances in that hierarchy level of the design. You can traverse down the hierarchy tree in this way to find the instance that contains the state machine. Click on the name of the state machine in the **State Machines** folder (Figure 12–32) to open the appropriate schematic in the schematic view (Figure 12–31).

Figure 12–32. State Machine Instance in RTL Viewer Hierarchy List



The State Machine Viewer opens (Figure 12–33).

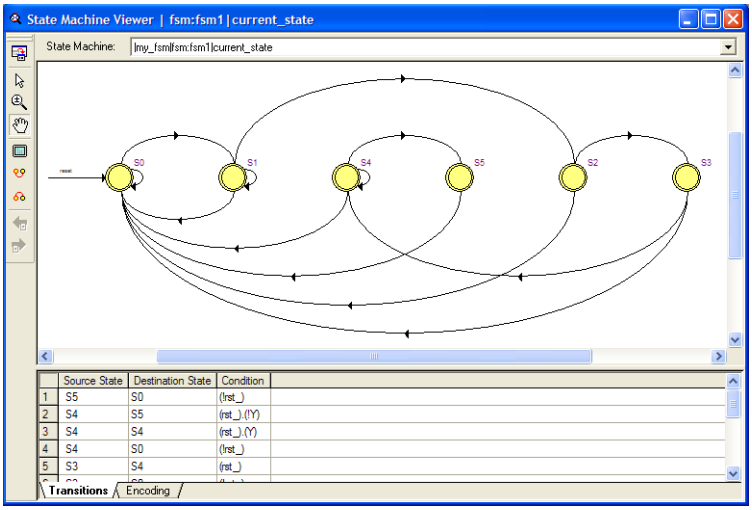
Figure 12–33. State Machine Viewer Showing Incorrect Transitions



2. You can analyze this state machine instance using the state machine diagram, transition table, and encoding table. Clearly something is wrong with the state machine because every state has a transition to every other state. After inspecting the state machine behavior, you determine that in this scenario, the designer forgot to create default assignments for the next state (that is, `next_state = current_state` if the conditions are not met).

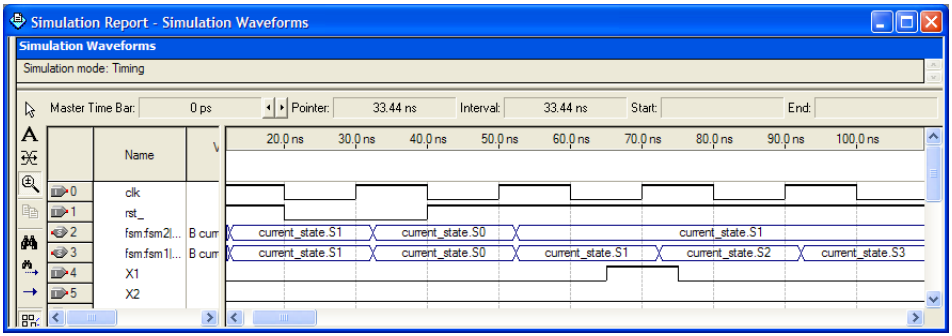
3. After fixing the error in the HDL code, recompile the design and repeat steps 1-2 to view the new state machine diagram and transition table (shown in Figure 12–34) and check that the state transitions now occur correctly.

Figure 12–34. State Machine Viewer Showing Correct Transitions



4. Perform a new simulation, as shown in Figure 12–35, and verify that the state machine now performs as expected.

Figure 12–35. Simulation Result Showing Correct States



Conclusion

The Quartus II RTL Viewer, State Machine Viewer, and Technology Map Viewer allow you to explore and analyze your initial synthesis netlist, post-synthesis netlist, or post-fitting and physical synthesis netlist. The viewers provide a number of features in the hierarchy list and schematic view to help you quickly trace through your netlist and find specific hierarchies or nodes of interest. These capabilities can help you debug, optimize, or constrain your design more efficiently to increase your productivity.

Document Revision History

Table 12–10 shows the revision history for this chapter.

Table 12–10. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No changes to content.	Updated for Quartus II software version 7.2.
May 2007 v7.1.0	• Renamed "Viewing the Properties of Instances and Primitives" on page 12–24 • Added "Viewing LUT Representations in the Technology Map Viewer" on page 12–24 • Renamed and updated "Customizing the Schematic Display in the RTL Viewer" on page 12–31 • Added "Grouping Combinational Logic into Logic Clouds" on page 12–32 • Added "Radial Menu" on page 12–50 • Updated Table 12–1 • Updated Table 12–4 • Updated Table 12–8 • Updated Figure 12–7 • Updated Figure 12–8	Chapter updated for Quartus II version 7.1.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Chapter 13 was formerly Chapter 12 in version 6.0.0. Updated for the Quartus II software version 6.1.0: • Added information about the Technology Map Viewer (Post-Mapping) • Can run the RTL Viewer as part of compilation flow, rather than wait for the Fitter to complete before viewing the netlist • Customized the schematic display for better viewing and to speed up the debugging process • Added support for Stratix III devices	With the addition of the Technology Map Viewer (Post-Mapping), you can view both the post-mapping and post-fitting netlists at the same time. Other changes also speed up the debugging process.

Table 12–10. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
May 2006 v6.0.0	• Name changed to *Analyzing Designs with the Quartus II Netlist Viewers*. • Updated for the Quartus II software version 6.0: • Updated GUI information.	—
December 2005 v5.1.1	Updated for version 5.1, including viewing inside device atoms, filter on bus index, display timing path in the RTL Viewer, state machine access from Tools menu, locate from state machines, and state encoding table.	—
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 12 was formerly chapter 14 in version 5.0.	—
May 2005 v5.0.0	Chapter 14 was formerly chapter 12 in version 4.2.	—
December 2004 v2.1	• Chapter 13 was formerly Chapter 14 in version 4.1. • Updates to tables and figures. • New functionality for Quartus II software version 4.2.	—
June 2004 v 2.0	• Updates to tables, and figures. • New functionality for Quartus II software version 4.1.	—
February 2004 v1.0	Initial release.	—



Quartus II Version 7.2 Handbook

Volume 2: Design Implementation and Optimization



101 Innovation Drive
San Jose, CA 95134
www.altera.com

QII5V2-7.2

Copyright © 2007 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

I.S. EN ISO 9001

Contents

Chapter Revision Dates ..... **xv**

About this Handbook ..... **xvii**
How to Contact Altera xvii
Third-Party Software Product Information xvii
Typographic Conventions xviii

Section I. Scripting and Constraint Entry

Chapter 1. Assignment Editor
Introduction 1–1
Using the Assignment Editor 1–1
Category, Node Filter, Information and Edit Bars 1–2
Viewing and Saving Assignments in the Assignment Editor 1–6
Assignment Editor Features 1–7
Using the Enhanced Spreadsheet Interface 1–8
Dynamic Syntax Checking 1–9
Node Filter Bar 1–10
Using Assignment Groups 1–11
Customizable Columns 1–12
Tcl Interface 1–13
Assigning Pin Locations Using the Assignment Editor 1–14
Creating Timing Constraints Using the Assignment Editor 1–14
Exporting and Importing Assignments 1–15
Exporting Assignments 1–16
Exporting Pin Assignments 1–16
Importing Assignments 1–18
Conclusion 1–20
Referenced Documents 1–20
Document Revision History 1–21

Chapter 2. Command-Line Scripting
Introduction 2–1
The Benefits of Command-Line Executables 2–1
Introductory Example 2–2
Command-Line Executables 2–3
Command-Line Scripting Help 2–6
Command-Line Option Details 2–7

Option Precedence 2–8
Design Flow 2–11
Compilation with quartus_sh --flow 2–11
Text-Based Report Files 2–12
Makefile Implementation 2–13
Command-Line Scripting Examples 2–16
Create a Project and Apply Constraints 2–16
Check Design File Syntax 2–18
Create a Project and Synthesize a Netlist Using Netlist Optimizations 2–19
Archive and Restore Projects 2–19
Perform I/O Assignment Analysis 2–20
Update Memory Contents without Recompiling 2–20
Fit a Design as Quickly as Possible 2–21
Fit a Design Using Multiple Seeds 2–21
The QFlow Script 2–23
Referenced Documents 2–23
Document Revision History 2–24

Chapter 3. Tcl Scripting

Introduction 3–1
What is Tcl? 3–2
Quartus II Tcl Packages 3–2
Loading Packages 3–5
Quartus II Tcl API Help 3–5
Executables Supporting Tcl 3–9
Command-Line Options: -t, -s, and --tcl_eval 3–10
Using the Quartus II Tcl Console Window 3–11
End-to-End Design Flows 3–12
Creating Projects and Making Assignments 3–13
EDA Tool Assignments 3–14
Using LogicLock Regions 3–18
Compiling Designs 3–21
Reporting 3–22
Creating CSV Files for Excel 3–24
Timing Analysis 3–25
Classic Timing Analysis 3–26
TimeQuest Timing Analysis 3–29
Automating Script Execution 3–30
Making the Assignment 3–31
Script Execution 3–31
Execution Example 3–32
Controlling Processing 3–33
Displaying Messages 3–33
Other Scripting Features 3–33
Natural Bus Naming 3–33
Using Collection Commands 3–34
Using the post_message Command 3–35

Accessing Command-Line Arguments 3–36
Using the Quartus II Tcl Shell in Interactive Mode 3–39
Quartus II Legacy Tcl Support 3–42
Tcl Scripting Basics 3–42
Hello World Example 3–42
Variables 3–43
Substitutions 3–43
Arithmetic 3–44
Lists 3–44
Arrays 3–45
Control Structures 3–46
Procedures 3–47
File I/O 3–48
Syntax and Comments 3–49
External References 3–50
Referenced Documents 3–50
Document Revision History 3–51

Chapter 4. Managing Quartus II Projects

Introduction 4–1
Creating a New Project 4–2
Using Revisions With Your Design 4–3
Creating and Deleting Revisions 4–3
Comparing Revisions 4–6
Creating Different Versions of Your Design 4–7
Archiving Projects with the Quartus II Archive Project Feature 4–8
Version-Compatible Databases 4–10
Quartus II Project Platform Migration 4–11
Filenames and Hierarchy 4–11
Search Path Precedence Rules 4–15
Quartus II-Generated Files for Third-Party EDA Tools 4–15
Migrating Database Files 4–15
Working with Messages 4–16
Messages Window 4–17
Hiding Messages 4–18
Message Suppression 4–19
Message Suppression Methods 4–21
Details and Limitations 4–21
Message Suppression Manager 4–22
Quartus II Settings File 4–25
Format Preservation 4–25
Quartus II Default Settings File 4–26
Scripting Support 4–26
Managing Revisions 4–27
Archiving Projects with a Tcl Command or at the Command Prompt 4–28
Restoring Archived Projects 4–28
Importing and Exporting Version-Compatible Databases 4–28

Specifying Libraries Using Scripts 4–29
Conclusion 4–30
Referenced Documents 4–30
Document Revision History 4–31

Section II. I/O and PCB Tools

Chapter 5. I/O Management

Introduction 5–1
Understanding Altera FPGA Pin Terminology 5–2
Package Pins 5–2
Pads 5–3
I/O Banks 5–4
VREF Groups 5–6
Importing and Exporting Pin Assignments 5–6
CSV File 5–6
Quartus II Settings Files (QSFs) 5–7
Tcl Script 5–7
FPGA Xchange File 5–8
PIN File 5–9
Create a Megafunction or IP MegaCore Variation from the Pin Planner 5–14
Import a Megafunction or IP MegaCore Variation from the Pin Planner 5–14
Create a Top-Level Design File for I/O Analysis 5–15
Creating Pin-Related Assignments 5–21
Using the Pin Planner 5–22
Assignment Editor 5–57
Tcl Scripts 5–61
Chip Planner or Timing Closure Floorplan 5–62
Synthesis Attributes 5–63
Validating Pin Assignments 5–65
Using the Live I/O Check Feature to Validate Pin Assignments 5–65
Using I/O Assignment Analysis to Validate Pin Assignments 5–67
Incorporating PCB Design Tools 5–87
Advanced I/O Timing 5–87
I/O Timing and Power with Capacitive Loading 5–88
Enabling and Configuring Advanced I/O Timing 5–90
Conclusion 5–98
Referenced Documents 5–98
Document Revision History 5–99

Chapter 6. Mentor Graphics PCB Design Tools Support

Introduction 6–1
FPGA-to-PCB Design Flow 6–2
Setting Up the Quartus II Software 6–5
Generating Pin-Out Files 6–7
Generating FPGA Xchange Files 6–7

Creating a Backup Quartus II Settings File 6–8
FPGA-to-Board Integration with the I/O Designer Software 6–8
I/O Designer Database Wizard 6–10
Updating Pin Assignments from the Quartus II Software 6–17
Sending Pin Assignment Changes to the Quartus II Software 6–20
Generating Symbols for the DxDesigner Software 6–22
Scripting Support 6–28
FPGA-to-Board Integration with the DxDesigner Software 6–30
DxDesigner Project Settings 6–30
DxDesigner Symbol Wizard 6–32
Conclusion 6–35
Referenced Documents 6–36
Document Revision History 6–36

Chapter 7. Cadence PCB Design Tools Support

Introduction 7–1
Product Comparison 7–2
FPGA-to-PCB Design Flow 7–3
Setting Up the Quartus II Software 7–5
Generating Pin-Out Files 7–6
FPGA-to-Board Integration with the Cadence Allegro Design Entry HDL Software 7–6
Symbol Creation 7–6
Instantiating the Symbol in the Cadence Allegro Design Entry HDL Software 7–17
FPGA-to-Board Integration with Allegro Design Entry CIS 7–18
Allegro Design Entry CIS Project Creation 7–19
Generate Part 7–19
Split Part 7–22
Instantiate Symbol in Design Entry CIS Schematic 7–24
Altera Libraries for Design Entry CIS 7–25
Conclusion 7–27
Referenced Document 7–27
Document Revision History 7–28

Section III. Area, Timing and Power Optimization

Chapter 8. Area and Timing Optimization

Introduction 8–1
Optimizing Your Design 8–2
Initial Compilation: Required Settings 8–3
Device Settings 8–3
I/O Assignments 8–4
Timing Requirement Settings 8–4
Device Migration Settings 8–7
Partitions and Floorplan Assignments for Incremental Compilation 8–7
Initial Compilation: Optional Settings 8–8

Design Assistant 8–9
Smart Compilation Setting 8–9
Early Timing Estimation 8–9
Optimize Hold Timing 8–10
Asynchronous Control Signal Recovery/Removal Analysis 8–11
Limit to One Fitting Attempt 8–12
Optimize Fast Corner Timing 8–12
Fitter Effort Setting 8–13
Design Analysis 8–14
Error and Warning Messages 8–15
Ignored Timing Assignments 8–15
Resource Utilization 8–15
I/O Timing (Including t_{PD}) 8–16
Register-to-Register Timing 8–19
Global Routing Resources 8–23
Compilation Time 8–24
Resource Utilization Optimization Techniques (LUT-Based Devices) 8–25
Using the Resource Optimization Advisor 8–25
Resolving Resource Utilization Issues Summary 8–26
I/O Pin Utilization or Placement 8–26
Logic Utilization or Placement 8–27
Routing 8–38
Timing Optimization Techniques (LUT-Based Devices) 8–42
Timing Optimization Advisor 8–42
I/O Timing Optimization 8–43
Register-to-Register Timing Optimization Techniques (LUT-Based Devices) 8–52
LogicLock Assignments 8–63
Location Assignments and Back-Annotation 8–67
Resource Utilization Optimization Techniques (Macrocell-Based CPLDs) 8–71
Use Dedicated Inputs for Global Control Signals 8–71
Reserve Device Resources 8–72
Pin Assignment Guidelines and Procedures 8–72
Resolving Resource Utilization Problems 8–75
Timing Optimization Techniques (Macrocell-Based CPLDs) 8–79
Improving Setup Time 8–80
Improving Clock-to-Output Time 8–80
Improving Propagation Delay (tPD) 8–82
Improving Maximum Frequency (fMAX) 8–82
Optimizing Source Code—Pipelining for Complex Register Logic 8–83
Compilation-Time Optimization Techniques 8–86
Incremental Compilation 8–86
Use Multiple Processors for Parallel Compilation 8–87
Reduce Synthesis Time and Synthesis Netlist Optimization Time 8–88
Check Early Timing Estimation before Fitting 8–89
Reduce Placement Time 8–89
Reduce Routing Time 8–92
Other Optimizing Resources 8–93
Design Space Explorer 8–93

Power Optimization Advisor 8–93
Scripting Support 8–94
Initial Compilation Settings 8–95
Resource Utilization Optimization Techniques (LUT-Based Devices) 8–96
I/O Timing Optimization Techniques (LUT-Based Devices) 8–97
Register-to-Register Timing Optimization Techniques (LUT-Based Devices) 8–97
Conclusion 8–98
Referenced Documents 8–99
Document Revision History 8–100

Chapter 9. Power Optimization

Introduction 9–1
Power Dissipation 9–2
Design Space Explorer 9–3
Power-Driven Compilation 9–5
Power-Driven Fitter 9–11
Recommended Flow for Power-Driven Compilation 9–15
Area-Driven Synthesis 9–15
Gate-Level Register Retiming 9–17
Design Guidelines 9–20
Clock Power Management 9–20
Reducing Memory Power Consumption 9–23
Pipelining and Retiming 9–26
Architectural Optimization 9–29
I/O Power Guidelines 9–33
Power Optimization Advisor 9–35
Conclusion 9–38
Referenced Documents 9–39
Document Revision History 9–40

Chapter 10. Analyzing and Optimizing the Design Floorplan

Introduction 10–1
Chip Planner Overview 10–2
Starting Chip Planner 10–3
Chip Planner Toolbar 10–3
Chip Planner Tasks and Layers 10–4
LogicLock Regions 10–6
Creating LogicLock Regions 10–7
Placing LogicLock Regions 10–8
Placing Device Features into LogicLock Regions 10–8
LogicLock Regions Window 10–8
Excluded Resources 10–10
Hierarchical (Parent and Child) LogicLock Regions 10–10
Using LogicLock Regions in the Chip Planner 10–11
Assigning LogicLock Region Content 10–11
Creating LogicLock Regions with the Chip Planner 10–12
Viewing Connections Between LogicLock Regions in the Chip Planner 10–12

Design Analysis Using the Chip Planner 10–13
Chip Planner Floorplan Views 10–13
Viewing Architecture-Specific Design Information 10–19
Viewing Critical Paths 10–20
Viewing Physical Timing Estimates 10–22
Viewing Routing Congestion 10–23
Viewing I/O Banks 10–24
Generating Fan-In and Fan-Out Connections 10–26
Generating Immediate Fan-In and Fan-Out Connections 10–26
Highlight Routing 10–28
Show Delays 10–29
Exploring Paths in the Chip Planner 10–30
Viewing Assignments in the Chip Planner 10–32
Viewing Routing Channels for a Path in the Chip Planner 10–33
Cell Delay Table 10–34
Viewing High and Low Power Tiles in Stratix III Devices in the Chip Planner 10–35
Timing Closure Floorplan Overview 10–36
Design Analysis Using the Timing Closure Floorplan 10–38
Timing Closure Floorplan Views 10–38
Viewing Assignments 10–40
Viewing Critical Paths 10–42
Physical Timing Estimates 10–44
Viewing Routing Congestion 10–47
Timing Closure Floorplan View 10–48
LogicLock Regions in the Timing Closure Floorplan 10–49
Analyzing LogicLock Region Connectivity Using the Timing Closure Floorplan 10–50
Additional Quartus II LogicLock Design Features 10–52
Using LogicLock Methodology for Older Device Families 10–57
The Quartus II LogicLock Methodology 10–58
Improving Design Performance 10–59
LogicLock Restrictions 10–59
Preserving Timing Results Using the LogicLock Flow 10–60
Importing and Exporting LogicLock Regions 10–61
Back-Annotating Routing Information 10–65
Back-Annotating LogicLock Regions 10–65
Exporting Back-Annotated Routing in LogicLock Regions 10–66
Importing Back-Annotated Routing in LogicLock Regions 10–67
Scripting Support 10–67
Initializing and Uninitializing a LogicLock Region 10–68
Creating or Modifying LogicLock Regions 10–68
Obtaining LogicLock Region Properties 10–69
Assigning LogicLock Region Content 10–69
Prevent Further Netlist Optimization 10–69
Save a Node-Level Netlist for the Entire Design into a Persistent Source File 10–70
Exporting LogicLock Regions 10–70
Importing LogicLock Regions 10–70
Setting LogicLock Assignment Priority 10–71

Assigning Virtual Pins 10–71
Back-Annotating LogicLock Regions 10–71
Conclusion 10–72
Referenced Documents 10–73
Document Revision History 10–73

Chapter 11. Netlist Optimizations and Physical Synthesis

Introduction 11–1
Synthesis Netlist Optimizations 11–3
WYSIWYG Primitive Resynthesis 11–3
Gate-Level Register Retiming 11–5
Preserving Synthesis Netlist Optimization Results 11–10
Physical Synthesis Optimizations 11–11
Automatic Asynchronous Signal Pipelining 11–13
Physical Synthesis for Combinational Logic 11–14
Physical Synthesis for Registers—Register Duplication 11–15
Physical Synthesis for Registers—Register Retiming 11–16
Preserving Your Physical Synthesis Results 11–17
Applying Netlist Optimization Options 11–19
Scripting Support 11–20
Synthesis Netlist Optimizations 11–20
Physical Synthesis Optimizations 11–21
Incremental Compilation 11–22
Back-Annotating Assignments 11–22
Conclusion 11–23
Referenced Documents 11–23
Document Revision History 11–23

Chapter 12. Design Space Explorer

Introduction 12–1
DSE Concepts 12–1
DSE Exploration 12–2
General Description 12–2
Timing Analyzer Support 12–4
DSE Flow 12–5
DSE Support for Altera Device Families 12–6
DSE Project Settings 12–7
Setting Up the DSE Work Environment 12–7
Specifying the Revision 12–7
Setting the Initial Seed 12–7
Restructuring LogicLock Regions 12–7
Quartus II Integrated Synthesis 12–9
Performing an Advanced Search in Design Space Explorer 12–9
Exploration Space 12–10
Optimization Goal 12–13
Quality of Fit (QoF) 12–14
Search Method 12–15

DSE Flow Options 12–15
Create a Revision from a DSE Point 12–15
Change Decision Column 12–16
Stop Flow When Zero Failing Paths are Achieved 12–17
Continue Exploration Even If Base Compilation Fails 12–17
Run Quartus II PowerPlay Power Analyzer During Exploration 12–17
Archive All Compilations 12–18
Stop Flow After Time 12–18
Save Exploration Space to File 12–18
Ignore SignalTap and SignalProbe Settings 12–18
Skip Base Analysis and Compilation If Possible 12–18
Lower Priority of Compilation Threads 12–19
DSE Configuration File 12–19
DSE Advanced Information 12–20
Computer Load Sharing in DSE Using Distributed Exploration 12–20
Concurrent Local Compilations 12–22
Creating Custom Spaces for DSE 12–22
Referenced Documents 12–27
Document Revision History 12–27

Chapter 13. Synplicity Amplify Physical Synthesis Support

Introduction 13–1
Software Requirements 13–1
Amplify Physical Synthesis Concepts 13–2
Amplify-to-Quartus II Flow 13–3
Initial Pass: No Physical Constraints 13–3
Iterative Passes: Optimizing the Critical Paths 13–5
Using the Amplify Physical Optimizer Floorplans 13–6
Multiplexers 13–7
Independent Paths 13–9
Feedback Paths 13–9
Starting and Ending Points 13–9
Utilization 13–11
Detailed Floorplans 13–11
Forward Annotating Amplify Physical Optimizer Constraints into the Quartus II Software 13–12
Altera Megafunctions Using the MegaWizard Plug-In Manager with the Amplify Software 13–13
Conclusion 13–14
Referenced Documents 13–14
Document Revision History 13–15

Section IV. Engineering Change Management

Chapter 14. Engineering Change Management with the Chip Planner

Introduction 14–1
Engineering Change Orders 14–2
Performance 14–2
Compilation Time 14–3
Verification 14–3
Documentation 14–4
ECO Design Flow 14–5
The Chip Planner Overview 14–6
Opening the Chip Planner 14–7
The Chip Planner Toolbar 14–7
The Chip Planner Tasks and Layers 14–8
The Chip Planner Floorplan Views 14–9
First-Level View 14–10
Second-Level View 14–11
Third-Level View 14–12
Bird's Eye View 14–13
Performing ECOs with the Chip Planner (Floorplan View) 14–15
Creating Atoms 14–15
Deleting Atoms 14–20
Moving Atoms 14–20
Check and Save Netlist changes 14–21
Resource Property Editor 14–21
Logic Element 14–21
Adaptive Logic Module 14–25
FPGA I/O Elements 14–27
FPGA RAM Blocks 14–33
FPGA DSP Blocks 14–34
Change Manager 14–36
Complex Changes in the Change Manager 14–37
Managing SignalProbe Signals 14–38
Exporting Changes 14–38
Using Incremental Compilation in the ECO Flow 14–38
ECO Flow with No Quartus II Incremental Compilation 14–40
Scripting Support 14–40
Common ECO Applications 14–40
Adjust the Drive Strength of an I/O Using the Chip Planner 14–41
Modifying the PLL Properties Using the Chip Planner 14–42
PLL Properties 14–43
Performing Static Timing Analysis 14–46
Generating a Programming File 14–46
Conclusion 14–46
Referenced Documents 14–47
Document Revision History 14–47

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook, Volume 2*, were revised on the following dates. Where chapters are available separately, part numbers are listed.

Chapter 1. Assignment Editor
Revised: *October 2007*
Part number: *QII52001-7.2.0*

Chapter 2. Command-Line Scripting
Revised: *October 2007*
Part number: *QII52002-7.2.0*

Chapter 3. Tcl Scripting
Revised: *October 2007*
Part number: *QII52003-7.2.0*

Chapter 4. Managing Quartus II Projects
Revised: *October 2007*
Part number: *QII52012-7.2.0*

Chapter 5. I/O Management
Revised: *October 2007*
Part number: *QII52013-7.2.0*

Chapter 6. Mentor Graphics PCB Design Tools Support
Revised: *October 2007*
Part number: *QII52015-7.2.0*

Chapter 7. Cadence PCB Design Tools Support
Revised: *October 2007*
Part number: *QII52014-7.2.0*

Chapter 8. Area and Timing Optimization
Revised: *October 2007*
Part number: *QII52005-7.2.0*

Chapter 9. Power Optimization
Revised: *October 2007*
Part number: *QII52016-7.2.0*

Chapter 10. Analyzing and Optimizing the Design Floorplan
Revised: *October 2007*
Part number: *QII52006-7.2.0*

Chapter 11. Netlist Optimizations and Physical Synthesis
Revised: *October 2007*
Part number: *QII52007-7.2.0*

Chapter 12. Design Space Explorer
Revised: *October 2007*
Part number: *QII52008-7.2.0*

Chapter 13. Synplicity Amplify Physical Synthesis Support
Revised: *October 2007*
Part number: *QII52011-7.2.0*

Chapter 14. Engineering Change Management with the Chip Planner
Revised: *October 2007*
Part number: *QII52017-7.2.0*

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 7.2.

How to Contact Altera

For the most up-to-date information about Altera products, refer to the following table.

Information Type	Contact *(1)*
Technical support	www.altera.com/mysupport
Technical training	www.altera.com/training custrain@altera.com
Product literature	www.altera.com/literature
Altera literature services	literature@altera.com *(1)*
FTP site	ftp.altera.com

Note to table:
(1) You can also contact your local Altera sales office or sales representative.

Third-Party Software Product Information

Third-party software products described in this handbook are not Altera products, are licensed by Altera from third parties, and are subject to change without notice. Updates to these third-party software products may not be concurrent with Quartus II software releases. Altera has assumed responsibility for the selection of such third-party software products and its use in the Quartus II 7.2 software release. To the extent that the software products described in this handbook are derived from third-party software, no third party warrants the software, assumes any liability regarding use of the software, or undertakes to furnish you any support or information relating to the software. EXCEPT AS EXPRESSLY SET FORTH IN THE APPLICABLE ALTERA PROGRAM LICENSE SUBSCRIPTION AGREEMENT UNDER WHICH THIS SOFTWARE WAS PROVDED TO YOU, ALTERA AND THIRD-PARTY LICENSORS DISCLAIM ALL WARRANTIES WITH RESPECT TO THE USE OF SUCH THIRD-PARTY SOFTWARE CODE OR DOCUMENTATION IN THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT. For more information, including the latest available version of specific third-party software products, refer to the documentation for the software in question.

Typographic Conventions

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
✓, —, N/A	Used in table cells to indicate the following: ✓ indicates a "Yes" or "Applicable" statement; — indicates a "No" or "Not Supported" statement; N/A indicates that the table cell entry is not applicable to the item of interest.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
	The hand points to information that requires special attention.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information about a particular topic.

Section I. Scripting and Constraint Entry

As a result of the increasing complexity of today's FPGA designs and the demand for higher performance, designers must make a large number of complex timing and logic constraints to meet their performance requirements. Once you have created a project and your design, you can use the Quartus® II software Assignment Editor and other GUI features to specify your initial design constraints, such as pin assignments, device options, logic options, and timing constraints.

This section describes how to enter constraints in the Quartus II software, how to take advantage of Quartus II modular executables, and how to develop and run tool command language (Tcl) scripts to perform a wide range of functions, and how to manage the Quartus II project for your design.

This section includes the following chapters:

- Chapter 1, Assignment Editor
- Chapter 2, Command-Line Scripting
- Chapter 3, Tcl Scripting
- Chapter 4, Managing Quartus II Projects



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

1. Assignment Editor

QII52001-7.2.0

Introduction

The complexity of today's FPGA designs is compounded by the increasing density and associated pin counts of current FPGAs. It requires that you make a large number of pin assignments that include the pin locations and I/O standards to successfully implement a complex design in the latest generation of FPGAs.

To facilitate the process of entering these assignments, Altera® has developed an intuitive, spreadsheet interface called the Assignment Editor. The Assignment Editor is designed to make the process of creating, changing, and managing a large number of assignments as easy as possible.

This chapter discusses the following topics:

- "Using the Assignment Editor"
- "Assignment Editor Features" on page 1–7
- "Assigning Pin Locations Using the Assignment Editor" on page 1–14
- "Creating Timing Constraints Using the Assignment Editor" on page 1–14
- "Exporting and Importing Assignments" on page 1–15

Using the Assignment Editor

You can use the Assignment Editor throughout the design cycle. Before board layout begins, you can make pin assignments with the Assignment Editor. Throughout the design cycle, use the Assignment Editor to help achieve your design performance requirements by making timing assignments. You can also use the Assignment Editor to view, filter, and sort assignments based on node names or assignment type.

The Assignment Editor is a resizable window. This scalability makes it easy to view or edit your assignments right next to your design files. To open the Assignment Editor, click the **Assignment Editor** icon in the toolbar, or on the Assignments menu, click **Assignment Editor**.

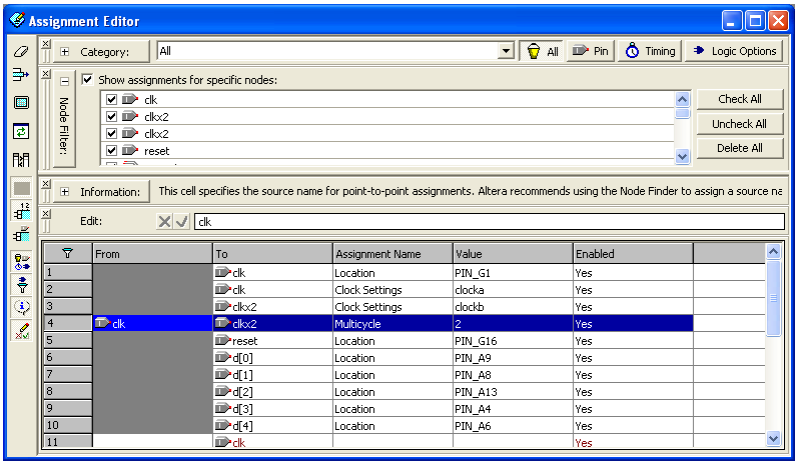


You can also launch the Assignment Editor by pressing Ctrl+Shift+A.

Category, Node Filter, Information and Edit Bars

The Assignment Editor window is divided into four bars and a spreadsheet (Figure 1–1).

Figure 1–1. The Assignment Editor Window



You can hide all four bars in the View menu if desired, and you can collapse the **Category**, **Node Filter**, and **Information** bars. Table 1–1 provides a brief description of each bar.

Table 1–1. Assignment Editor Bar Descriptions

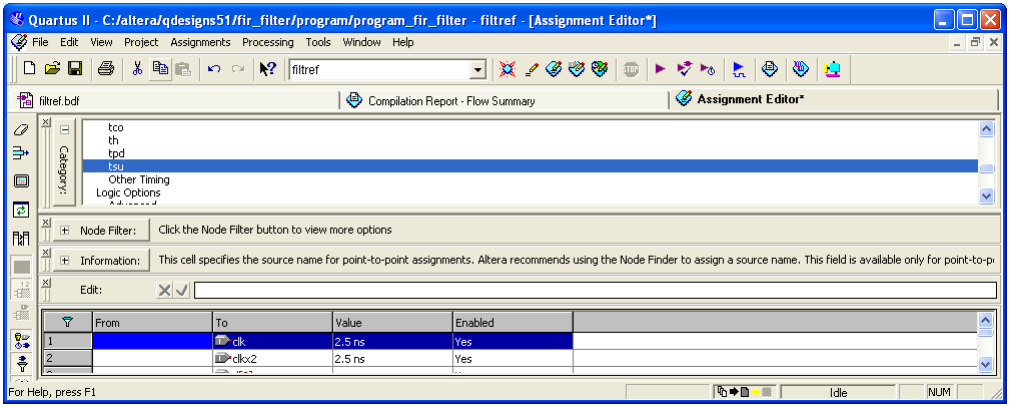
Bar Name	Description
Category	Lists the type of available assignments.
Node Filter	Lists a selection of design nodes to be viewed or assigned.
Information	Displays a description of the currently selected cell.
Edit	Allows you to edit the text in the currently selected cell(s).

Category Bar

The **Category** bar lists all assignment categories available for the selected device. You can use the **Category** bar to select a particular assignment type and to filter out all other assignments. Selecting an assignment category from the **Category** list changes the spreadsheet to show only applicable options and values. To search for a particular type of assignment, use the **Category** bar to filter out all other assignments.

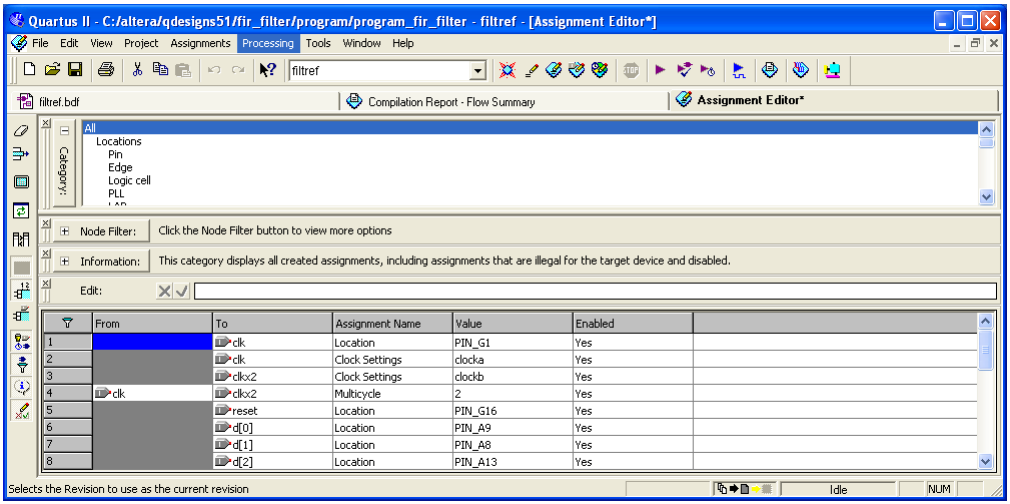
To view all t_{SU} assignments in your project, select **tsu** in the **Category** list (Figure 1–2).

Figure 1–2. t_{SU} Selected in the Category List



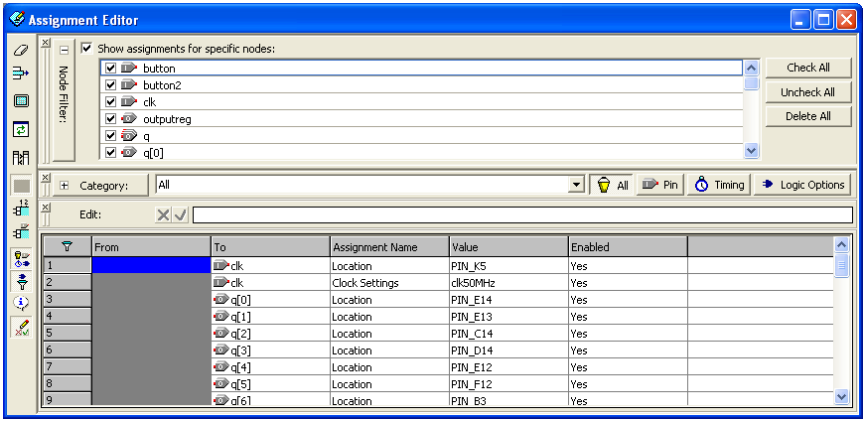
If you select **All** in the **Category** bar (Figure 1–3), the Assignment Editor displays all assignments.

Figure 1–3. All Selected in the Category List



When you collapse the **Category** bar, four shortcut buttons are displayed allowing you to select from various preset categories (Figure 1–4).

Figure 1–4. Category Bar



Use the **Pin** category to create pin location assignments. The **Pin** category displays additional information about each FPGA pin including its I/O Bank number, VREF group number, corresponding pad number, and primary and secondary functions.

When entering a pin number, the Assignment Editor auto completes the pin number. For example, instead of typing `Pin_AA3`, you can type `AA3` and let the Assignment Editor auto complete the pin number to `Pin_AA3`. You can also choose a pin location from the pins list by double clicking the cell in the location column. All occupied pin locations are shown in italics.

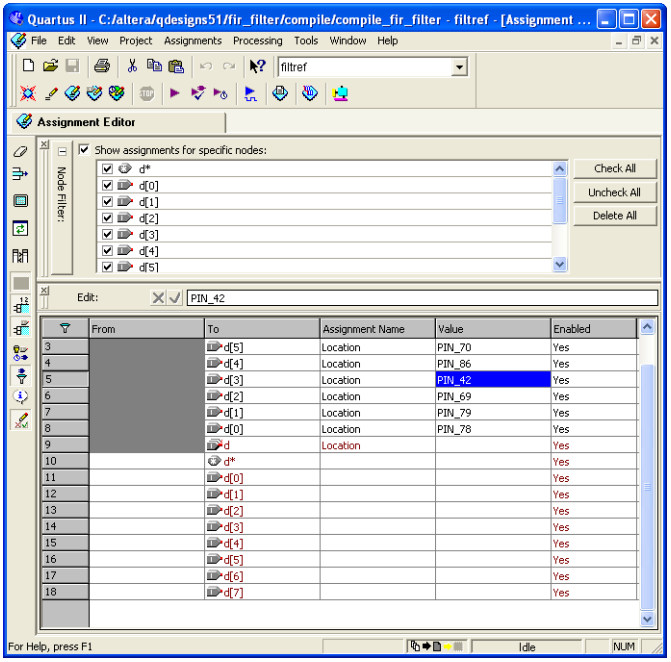
Node Filter Bar

When **Show assignments for specific nodes** is turned on, the spreadsheet shows only assignments for nodes matching the selected node name filters in the **Node Filter** bar. You can selectively enable individual node name filters listed in the **Node Filter** bar. You can create a new node name filter by selecting a node name with the Node Finder or typing a new node name filter. The Assignment Editor automatically inserts a spreadsheet row and pre-populates the **To** field with the node name filter. You can easily add an assignment to the matching nodes by entering it in the new row. Rows with incomplete assignments are shown in dark red. When you choose **Save** on the File menu, and there are incomplete assignments, a prompt gives you the choice to save and lose incomplete assignments, or cancel the save.

As shown in Figure 1–5, when all the bits of the d input bus are enabled in the **Node Filter** bar, all unrelated assignments are filtered out.

In the **Node Filter** bar, selecting a d input bus only highlights the row. If you want to enable the bus, you must turn on the bus.

Figure 1–5. Using the Node Filter Bar in the Assignment Editor



Information Bar

The **Information** bar provides a brief description of the currently selected cell and what information you should enter into the cell. For example, the **Information** bar describes if it is correct to enter a node name, or a number value into a cell. If the selected cell is a logic option, then the **Information** bar shows a description of that option.

For more information on logic options, refer to the Quartus® II Help.

Edit Bar

The **Edit** bar is an efficient way to enter a value into one or more spreadsheet cells.

To change the contents of multiple cells at the same time, select the cells in the spreadsheet (Figure 1–6), then type the new value into the **Edit** box in the **Edit** bar, and click **Accept** (Figure 1–7).

Figure 1–6. Edit Bar Selection

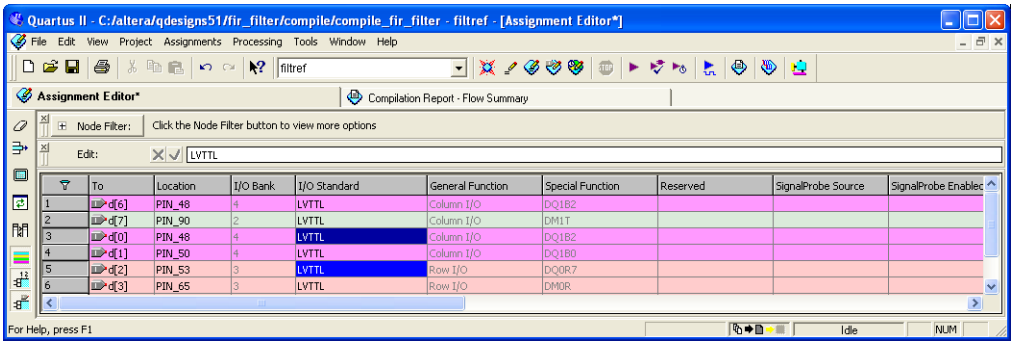
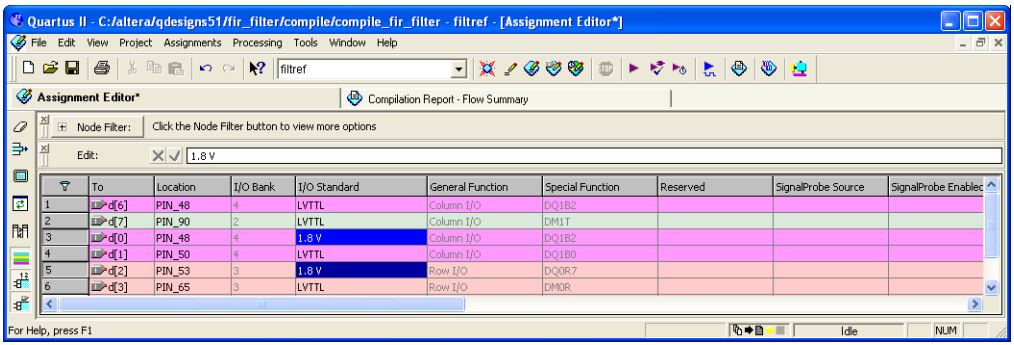


Figure 1–7. Edit Bar Change



Viewing and Saving Assignments in the Assignment Editor

Although the Assignment Editor is the most common method of entering and modifying assignments, there are other methods you can use to make and edit assignments. For this reason, you can refresh the Assignment Editor after you add, remove, or change an assignment outside the Assignment Editor.

By default, all assignments made in the Quartus II software are first stored into memory, then to the Quartus II Setting File (**.qsf**) on the disk after you start a processing task, or if you save or close your project. Saving assignments to memory avoids reading and writing to your disk drive and improves the performance of the software.

After making assignments in the Assignment Editor, on the File menu, click **Save** to save your assignments and update the Quartus II Settings File outside the Assignment Editor.

Starting with the Quartus II software version 5.1, you can force all assignments to be written to a disk drive. This is performed by turning off **Update assignments to disk during design processing only** in the **Processing** page of the **Options** settings dialog box on the Tools menu.



For more information on how the Quartus II software writes to the Quartus II Settings File, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

You can refresh the Assignment Editor window by clicking **Refresh** from the View menu. If you make an assignment in the Quartus II software, such as in the Tcl console or in the Pin Planner, the Assignment Editor reloads the new assignments from memory. If you directly modify the Quartus II Settings File outside the Assignment Editor, click **Refresh** on the View menu to view the assignments.



If the Quartus II Settings File is edited while the project is open, go to the File menu and click **Save Project** to ensure that you are editing the latest Quartus II Settings File.

Each time the Assignment Editor is refreshed, the following message displays in the Message window:

```
Info: Assignments reloaded -- assignments updated outside Assignment Editor
```

Assignment Editor Features

You can open the Assignment Editor from many locations in the Quartus II software, including the Text Editor, the Node Finder, the Timing Closure Floorplan, the Pin Planner, the Compilation Report, and the Messages window. For example, you can highlight a node name in your design file and open the Assignment Editor with the node name populated.

You can also open other windows from the Assignment Editor. From a node listed in the Assignment Editor spreadsheet, you can locate the node in any of the following windows: Pin Planner, Timing Closure Floorplan, Chip Planner, Block Editor, or Text Editor.

Using the Enhanced Spreadsheet Interface

One of the key features of the Assignment Editor is the spreadsheet interface. With the spreadsheet interface, you can sort columns, use pull-down list boxes, and copy and paste multiple cells into the Assignment Editor. As you enter an assignment, the font color of the row changes to indicate the status of the assignment. Refer to "Dynamic Syntax Checking" on page 1–9 for more information.

There are many ways to select or enter nodes into the spreadsheet including: the Node Finder, the Node Filter bar, the Edit bar, or by directly typing the node name into the cell in the spreadsheet. A node type icon is shown beside each node name and node name filter to identify its type. The node type icon identifies the entry as an input, output, bidirectional pin, a register, combinational logic, or an assignment group (Figure 1–8). The node type icon appears as an asterisk for node names and node name filters that use a wildcard character (* or ?).

Figure 1–8. Node Type Icon Displayed Beside Each Node Name in the Spreadsheet

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reserved	SignalProbe Source
1	follow	PIN_68	3	LVTTL	Row I/O	VREF1B3		
2	d[6]	PIN_48	4	LVTTL	Column I/O	DQ1B2		
3	d[7]	PIN_90	2	LVTTL	Column I/O	DM1T		
4	d*	PIN_99	2	LVTTL	Column I/O	DEV_OE/DQ1T6		
5	inst4	PIN_98	2	LVTTL	Column I/O	DQ1T5		
6	inst5	PIN_57	3	LVTTL	Row I/O	VREF2B3		
7	inst5[0]	PIN_29	4	LVTTL	Column I/O	DQ1B5		

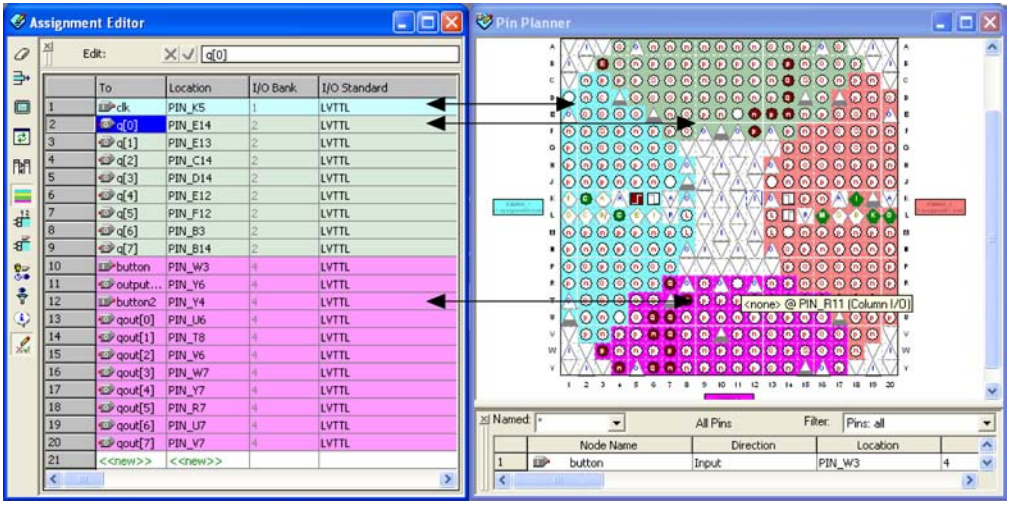
The Assignment Editor supports wildcards in the following types of assignments:

- All timing assignments
- Point-to-point global signal assignments (applicable to Stratix® II and Stratix devices)
- Point-to-point or pad-to-core delay chain assignments
- All assignments that support wild cards are shown in the drop list under the Assignment Name column of the Assignment Editor with "(Accepts wildcards/groups)" displayed beside it

The spreadsheet also supports customizable columns that allow you to show, hide, and arrange columns. For more information, refer to "Customizable Columns" on page 1–12.

When making pin location assignments, the background color of the cells coordinates with the color of the I/O bank shown in the Pin Planner (Figure 1–9).

Figure 1–9. Cells Coordinates with the Color of the I/O Bank



Dynamic Syntax Checking

As you enter your assignments, the Assignment Editor performs simple legality and syntax checks. This checking is not as thorough as the checks performed during compilation, but it rejects incorrect settings. For example, the Assignment Editor does not allow assignment of a pin name to a no-connect pin. In this case, the assignment is not accepted and you must enter a different pin location.

The color of the text in each row indicates if the assignment is incomplete, incorrect, or disabled (Table 1–2). To customize the colors in the Assignment Editor, on the Tools menu, click **Options**.

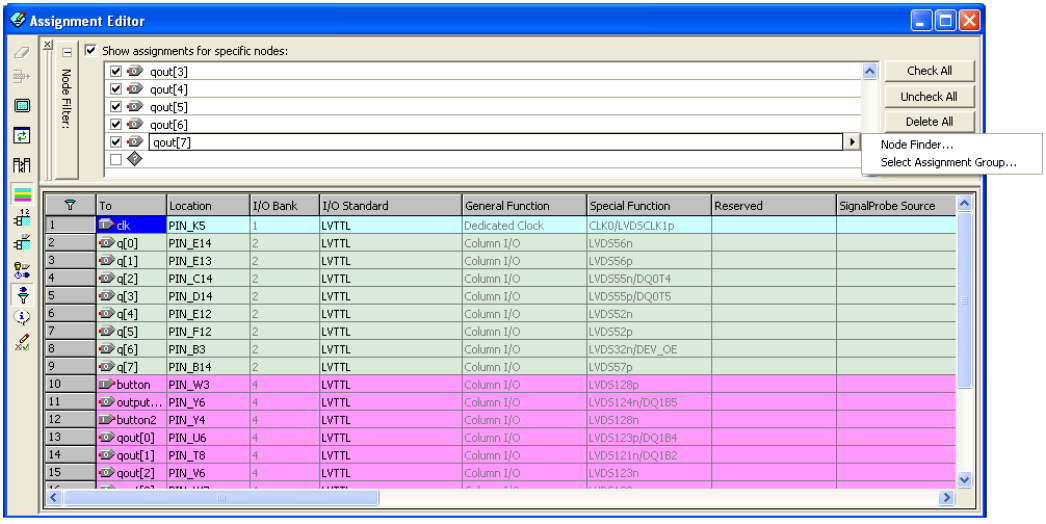
Table 1–2. Description of the Text Color in the Spreadsheet

Text Color	Description
Green	A new assignment can be created.
Yellow	The assignment contains warnings, such as an unknown node name.
Dark Red	The assignment is incomplete.
Bright Red	The assignment has an error, such as an illegal value.
Light Gray	The assignment is disabled.

Node Filter Bar

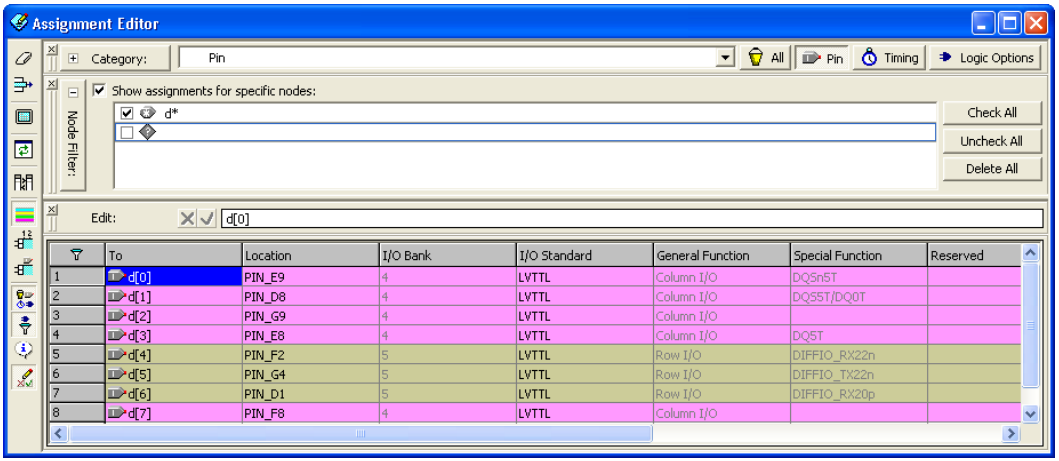
The **Node Filter** bar provides flexibility in how you view and make your settings. The **Node Filter** bar contains a list of node filters. To create a new entry, use the **Node Finder** or manually type the node name. Double-click an empty row in the **Node Filter** list, click on the **arrow**, and click **Node Finder** (Figure 1–10) to open the **Node Finder** dialog box.

Figure 1–10. Node Finder Option



In the **Node Filter** bar, you can turn each filter on or off. To turn off the **Node Filter** bar, turn off **Show assignments for specific nodes**. The wildcards (* and ?) are used to filter for a selection of all the design nodes with one entry in the Node Filter. For example, you can enter `d*` into the **Node Filter** list to view all assignments for `d[0]`, `d[1]`, `d[2]`, and `d[3]` (Figure 1–11).

Figure 1–11. Using the Node Filter Bar with Wildcards



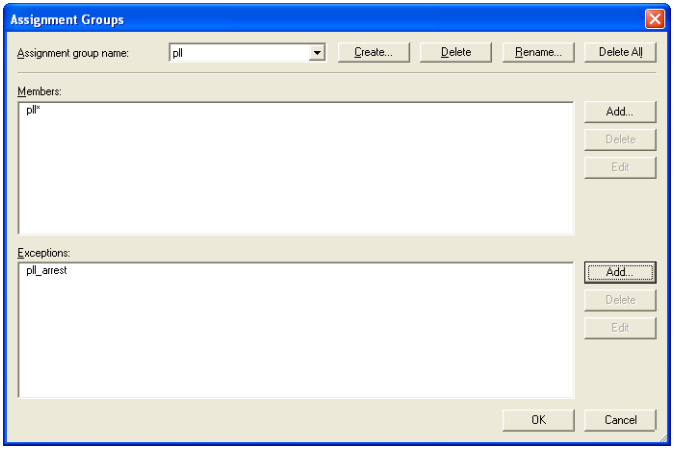
Using Assignment Groups

An assignment group is a collection of design nodes grouped together and represented as a single unit for the purpose of making assignments to the collection. Using assignment groups with the Assignment Editor provides the flexibility required for making complex fitting or timing assignments to a large number of nodes.

To create an assignment group, on the Assignments menu, click **Assignment (Time) Groups.** The **Assignment Groups** dialog box is shown. You can add or delete members of each assignment group with wild cards in the Node Finder (Figure 1–12).

For more information on using Assignment Groups for timing analysis, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Figure 1–12. Assignment Groups Dialog Box



There are cases when wildcards are not flexible enough to select a large number of nodes that have similar node names. You can use assignment groups to combine wildcards, which select a large number of nodes, and use exceptions to remove nodes that you did not intend to select. Although settings may not always display correctly when you have wildcards or assignment groups, the Fitter always recognizes assignments created with wildcards and assignment groups when the design is compiled.

Customizable Columns

To provide more control over the display of information in the spreadsheet, the Assignment Editor supports customizable columns.

You can move columns, sort them in ascending or descending order, show or hide individual columns, and align the content of the column left, center, or right for improved readability.

When the Quartus II software starts for the first time, you see a pre-selected set of columns. For example, when the Quartus II software is first started, the Comment column is hidden. To show or hide any of the available columns, on the View menu, click **Customize Columns**. When you restart the Quartus II software, your column settings are maintained.

Depending on the category selected, there are many different hidden columns you can display. For example, with the Pins category selected, there are many columns that are not shown by default, such as VREF group, pad number, output pin load, toggle rate, timing requirements, and fast input and output register options.

You can use the Comments column to document the purpose of a pin or to explain why you applied a timing or logic constraint. You can use the Enabled column to disable any assignment without deleting it. This feature is useful when performing multiple compilations with different timing constraints or logic optimizations.



Even though you can make many pin-related assignments with the **Pin** category selected, only the pin location assignment is disabled when you disable a row using the Enabled column.

Tcl Interface

Whether you use the Assignment Editor or another feature to create your design assignments, you can export them to a Tcl file. You can then use the Tcl file to reapply the settings or to archive your assignments. On the File menu, click **Export** to export your assignments (currently displayed in the spreadsheet of the Assignment Editor) to a Tcl script.



On the Project menu, click **Generate TCL File for Project** to generate a Tcl script file that sets up your design and applies all the assignments.

In addition, as you use the Assignment Editor to enter assignments, the equivalent Tcl commands are shown in the System Message window. You can reference these Tcl commands to create customized Tcl scripts (Figure 1–13). To copy a Tcl command from the Messages window, right-click the message and click **Copy**.

Figure 1–13. Equivalent Tcl Commands Displayed in the Messages Window





For more information on Tcl scripting with the Quartus II software, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Assigning Pin Locations Using the Assignment Editor

There are two methods for making pin assignments with the Assignment Editor. The first approach involves choosing a design node name for each device pin location. It is important to understand the properties of each pin on the FPGA device before you assign a design node to the location. For example, when following pin placement guidelines, you need to know which I/O bank or VREF group each pin belongs to.

On the Assignments menu, click **Assignment Editor**. To view all pin numbers in the targeted package, click the **Pin** category. On the View menu, click **Show All Assignable Pin Numbers**. You can customize the columns shown in the Assignment Editor to display property information about each pin including their pad numbers, as well as primary and secondary functions.



For more information on pin placement guidelines, refer to the *Selectable I/O Standards* chapters in the appropriate device handbook.

The second approach involves choosing a pin location for each pin in your design. To view all pin numbers in the targeted package, open the **Assignment Editor**, click the **Pin** category, and on the View menu, click **Show All Known Pin Names**. For each pin name, select a pin location.



For more information about creating pin assignments, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Creating Timing Constraints Using the Assignment Editor

Accurate timing constraints guide the place-and-route engine in the Quartus II software to help optimize your design into the FPGA. After completing a place-and-route, perform a static timing analysis using the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer to analyze slack and critical paths in your design.

If you are using the Quartus II Classic Timing Analyzer, create timing constraints using the Assignment Editor. On the Assignments menu, click **Assignment Editor**. In the **Category** list, select **Timing**, and make timing assignments in the spreadsheet section of the Assignment Editor.



For more information on the Quartus II Classic Timing Analyzer, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

If you are using the Quartus II TimeQuest Timing Analyzer, the TimeQuest Timing Analyzer uses timing assignments from a Synopsys Design Constraint (**.sdc**) file.



For information on converting the timing assignments in your Quartus Settings File to an Synopsys Design Constraint file, refer to the *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Exporting and Importing Assignments

Designs that use the LogicLock™ hierarchal design methodology use the **Import Assignment** command to import assignments into the current project. You can also use the **Export Assignments** command to save all the assignments in your project to a file to be used for archiving or to transfer assignments from one project to another.

On the Assignments menu, click **Export Assignments** or **Import Assignments** to do the following:

- Export your Quartus II assignments to a Quartus II Settings File.
- Import assignments from a Quartus II Entity Settings File (**.esf**), a MAX+PLUS® II Assignment and Configuration File (**.acf**), or a Comma Separated Value (**.csv**) file.

In addition to the **Export Assignments** and **Import Assignments** dialog boxes, the **Export** command on the File menu allows you to export your assignments to a Tcl Script (**.tcl**) file.



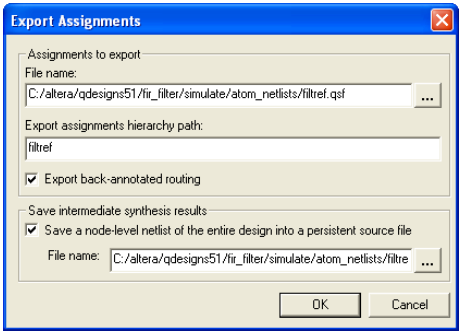
When applicable, the **Export** command exports the contents of the active window in the Quartus II software to another file format.

You can use these file formats for many different aspects of your project. For example, you can use a Comma Separated Value file for documentation purposes, or to transfer pin-related information to board layout tools. The Tcl file makes it easy to apply assignments in a scripted design flow. The LogicLock design flow uses the Quartus II Settings File to transfer your LogicLock region settings.

Exporting Assignments

You can use the **Export Assignments** dialog box to export your Quartus II software assignments into a Quartus II Settings File, generate a node-level netlist file, and export back-annotated routing information as a Routing Constraints File (**.rcf**) (Figure 1–14).

Figure 1–14. Export Assignments Dialog Box



On the Assignments menu, click **Export Assignments** to open the **Export Assignments** dialog box. The LogicLock design flow also uses this dialog box to export LogicLock regions.

For more information on using the **Export Assignments** dialog box to export LogicLock regions, refer to the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*.

On the File menu, click **Export** to export all assignments to a Tcl file or export a set of assignments to a Comma Separated Value file. When you export assignments to a Tcl file, only user-created assignments are written to the Tcl script file; default assignments are not exported.

When assignments are exported to a Comma Separated Value file, only the assignments displayed in the current view of the Assignment Editor are exported.

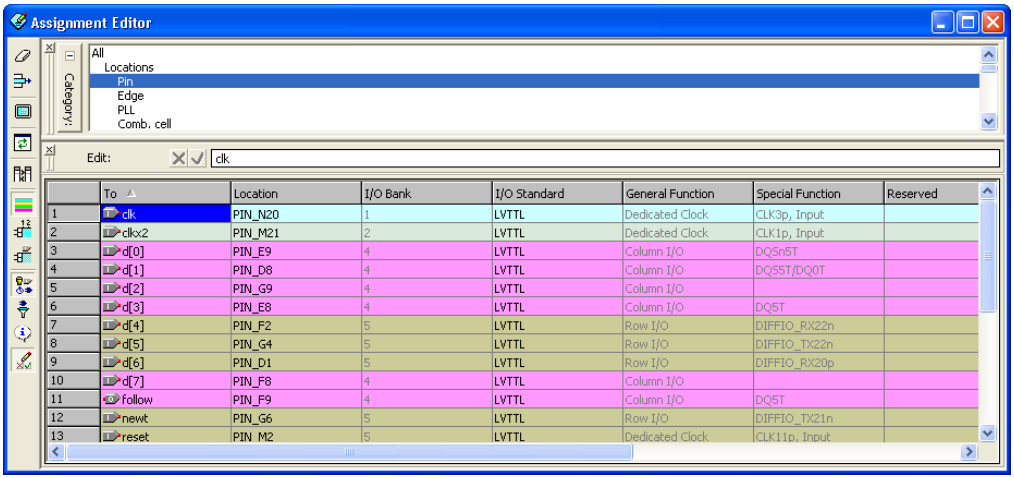
Exporting Pin Assignments

To export your pin assignments to a Comma Separated Value file, you can open the Assignment Editor and select **Pin** from the Category bar. The **Pin** category displays detailed properties about each pin similar to that of the device pin-out files in addition to the pin name and pin number. On the File menu, click **Export**, and select **Comma Separated Value File** from the **Save as type** list.

The first uncommented row of the Comma Separated Value file is a list of the column headings displayed in the Assignment Editor separated by commas. Each row below the header row represents the rows in the spreadsheet of the Assignment Editor (Figure 1–15). On the View menu, click **Customize Columns** to add and remove columns that are displayed in the spreadsheet. You can view and make edits to the Comma Separated Value file with Excel or other spreadsheet tools. If you intend to import the Comma Separated Value file back into the Quartus II software, the column headings must remain unedited and in the same order.

For more information on exporting pin assignments, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Figure 1–15. Assignment Editor With Category Set to Pin



The following code is an example of an exported Comma Separated Value file from the Assignment Editor:

```
# Note: The column header names should not be changed if you wish to import this .csv file
# into the Quartus II software.

To,Location,I/O Bank,I/O Standard,General Function,Special Function,Reserved,Enabled
clk,PIN_N20,1,LVTTL,Dedicated Clock,"CLK3p, Input",,Yes
clkx2,PIN_M21,2,LVTTL,Dedicated Clock,"CLK1p, Input",,Yes
d[0],PIN_E9,4,LVTTL,Column I/O,DQSn5T,,Yes
d[1],PIN_D8,4,LVTTL,Column I/O,DQS5T/DQ0T,,Yes
d[2],PIN_G9,4,LVTTL,Column I/O,,,Yes
d[3],PIN_E8,4,LVTTL,Column I/O,DQ5T,,Yes
d[4],PIN_F2,5,LVTTL,Row I/O,DIFFIO_RX22n,,Yes
d[5],PIN_G4,5,LVTTL,Row I/O,DIFFIO_TX22n,,Yes
d[6],PIN_D1,5,LVTTL,Row I/O,DIFFIO_RX20p,,Yes
d[7],PIN_F8,4,LVTTL,Column I/O,,,Yes
```

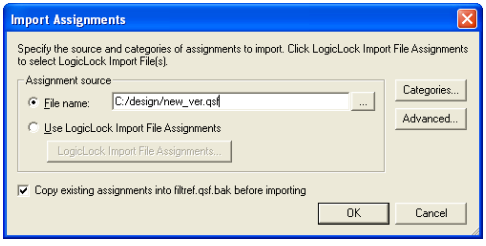
Importing Assignments

The **Import Assignments** dialog box allows you to import Quartus II assignments from a Quartus II Settings File, a Quartus II Entity Settings File, a MAX+PLUS II Assignment Configuration File, or a Comma Separated Value (Figure 1–16).

To import assignments from any of the supported assignment files, perform the following steps:

1. On the Assignments menu, click **Import Assignments**. The **Import Assignments** dialog box is shown (Figure 1–16).

Figure 1–16. Import Assignments Dialog Box



2. In the **File name** text-entry box, type the file name, or browse to the assignment file. The **Select File** dialog box is shown.

3. In the **Select File** dialog box, select the file, and click **Open**.

4. Click **OK**.



When you import a Comma Separated Value file, the first uncommented row of the file must be in the exact format as it was when exported.

When using the LogicLock flow methodology to import assignments, perform the following steps:

1. On the Assignments menu, click **Import Assignments**. The **Import Assignments** dialog box appears (Figure 1–16).
2. Turn on **Use LogicLock Import File Assignments,** and click **LogicLock Import File Assignments**.
3. When the **LogicLock Import File Assignments** dialog box opens, select the assignments to import and click **OK**.



For more information on using the **Import Assignments** dialog box to import LogicLock regions, refer to the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*.

You can create a backup copy of your assignments before importing new assignments by turning on the **Copy existing assignments into <*revision name*>.qsf.bak before importing** option.

When importing assignments from a file, you can choose which assignment categories to import by following these steps:

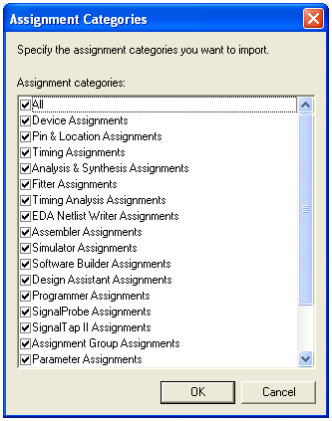
1. Click **Categories** in the **Import Assignments** dialog box.
2. Turn on the categories you want to import from the **Assignment categories** list (Figure 1–17).

To select specific types of assignments to import, click **Advanced** in the **Import Assignments** dialog box. The **Advanced Import Settings** dialog box appears. You can choose to import instance, entity, or global assignments, and select various assignment types to import.



For more information on these options, refer to the Quartus II Help.

Figure 1–17. Assignment Categories Dialog Box



Conclusion

As FPGAs continue to increase in density and pin count, it is essential to be able to quickly create and view design assignments. The Assignment Editor provides an intuitive and effective way of making assignments. With the spreadsheet interface and the **Category, Node Filter, Information**, and **Edit** bars, the Assignment Editor provides an efficient assignment entry solution for FPGA designers.

Referenced Documents

This chapter references the following documents:

- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*
- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Selectable I/O Standards* chapters in the appropriate device handbook
- *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 1–3 shows the revision history for this chapter.

Table 1–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 1–20.	Updated for the Quartus II software version 7.2.
May 2007 v7.1.0	Added Referenced Documents.	Updated the title of the referenced document *Quartus II Project Management* to *Managing Quartus II Projects.*
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Added revision history to document.	—
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0. • Added Quartus II Classic Timing Analyzer and Quartus II TimeQuest Timing Analyzer information.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.0.	—
May 2005 v5.0.0	• Updated for the Quartus II software version 5.0.0. • General formatting and editing updates. • Updated 2 graphics and references to reflect changes in the Quartus II software version 5.0.0	—
December 2004 v2.1	• Updated for Quartus II software version 4.2: • General formatting and editing updates. • Updated information about refreshing the Assignment Editor. • Updated figures. • Added information about how to make selections to the Assignment Editor window. • Added Time Groups reference. • Reworded description of Customizable Columns. • Added new section Creating Pin Locations Using the Assignment Editor. • Added new description to Exporting and Importing Assignments.	—
June 2004 v2.0	• Updates to tables, figures. • New functionality in the Quartus II software version 4.1.	—
February 2004 v1.0	Initial release.	—

2. Command-Line Scripting

QII52002-7.2.0

Introduction

FPGA design software that easily integrates into your design flow saves time and improves productivity. The Altera® Quartus® II software provides you with a command-line executable for each step of the FPGA design flow to make the design process customizable and flexible.

The benefits provided by command-line executables include:

- Command-line control over each step of the design flow
- Easy integration with scripted design flows including makefiles
- Reduced memory requirements
- Improved performance

The command-line executables are also completely compatible with the Quartus II GUI, allowing you to use the exact combination of tools that you prefer.

This chapter describes how to take advantage of Quartus II command-line executables, and provides several examples of scripts that automate different segments of the FPGA design flow.

The Benefits of Command-Line Executables

The Quartus II command-line executables provide command-line control over each step of the design flow. Each executable includes options to control commonly used software settings. Each executable also provides detailed, built-in help describing its function, available options, and settings.

Command-line executables allow for easy integration with scripted design flows. It is simple to create scripts in any language with a series of commands. These scripts can be batch-processed, allowing for integration with distributed computing in server farms. You can also integrate the Quartus II command-line executables in makefile-based design flows. All of these features enhance the ease of integration between the Quartus II software and other EDA synthesis, simulation, and verification software.

Command-line executables add integration and scripting flexibility without sacrificing the ease-of-use of the Quartus II GUI. You can use the Quartus II GUI and command-line executables at different stages in the design flow. For example, you might use the Quartus II GUI to edit the

floorplan for the design, use the command-line executables to perform place-and-route, and return to the Quartus II GUI to perform debugging with the Chip Editor.

Command-line executables reduce the amount of memory required during each step in the design flow. Because each executable targets only one step in the design flow, it is relatively compact, both in file size and the amount of memory used when running. This memory reduction improves performance, and is particularly beneficial in design environments where computer networks or workstations are heavily used with reduced memory.

Introductory Example

The following introduction to design flow with command-line executables shows how to create a project, fit the design, perform timing analysis, and generate programming files.

The tutorial design included with the Quartus II software is used to demonstrate this functionality. If installed, the tutorial design is found in the <*Quartus II directory*>**/qdesigns/fir_filter** directory.

Before making changes, copy the tutorial directory and type the four commands shown in Example 2–1 at a command prompt in the new project directory:



The <*Quartus II directory*>**/quartus/bin** directory must be in your PATH environment variable.

Example 2–1. Introductory Example

```
quartus_map filtref --source=filtref.bdf --family=CYCLONE ↵
quartus_fit filtref --part=EP1C12Q240C6 --fmax=80MHz --tsu=8ns ↵
quartus_asm filtref ↵
quartus_tan filtref ↵
```

The **quartus_map filtref --source=filtref.bdf --family=CYCLONE** command creates a new Quartus II project called **filtref** with the **filtref.bdf** file as the top-level file. It targets the Cyclone® device family and performs logic synthesis and technology mapping on the design files.

The **quartus_fit filtref --part=EP1C12Q240C6 --fmax=80MHz --tsu=8ns** command performs fitting on the **filtref** project. This command specifies an EP1C12Q240C6 device and the Fitter attempts to meet a global f_{MAX} requirement of 80 MHz and a global t_{SU} requirement of 8 ns.

The **quartus_asm filtref** command creates programming files for the **filtref** project.

The **quartus_tan filtref** command performs timing analysis on the **filtref** project to determine whether the design meets the timing requirements that were specified to the **quartus_fit** executable.

You can put the four commands from Example 2–1 into a batch file or script file, and run them. For example, you can create a simple UNIX shell script called **compile.sh**, which includes the code shown in Example 2–2.

Example 2–2. UNIX Shell Script: compile.sh

```
#!/bin/sh
PROJECT=filtref
TOP_LEVEL_FILE=filtref.bdf
FAMILY=Cyclone
PART=EP1C12Q240C6
FMAX=80MHz
quartus_map $PROJECT --source=$TOP_LEVEL_FILE --family=$FAMILY
quartus_fit $PROJECT --part=$PART --fmax=$FMAX
quartus_asm $PROJECT
quartus_tan $PROJECT
```

Edit the script as necessary and compile your project.

Command-Line Executables

Table 2–1 details the command-line executables and their respective descriptions.

Table 2–1. Quartus II Command-Line Executables and Descriptions (Part 1 of 4)

Executable	Description
Analysis and Synthesis quartus_map	Quartus II Analysis and Synthesis builds a single project database that integrates all the design files in a design entity or project hierarchy, performs logic synthesis to minimize the logic of the design, and performs technology mapping to implement the design logic using device resources such as logic elements.
Fitter quartus_fit	The Quartus II Fitter performs place-and-route by fitting the logic of a design into a device. The Fitter selects appropriate interconnection paths, pin assignments, and logic cell assignments. Quartus II Analysis and Synthesis must be run successfully before running the Fitter.

Table 2–1. Quartus II Command-Line Executables and Descriptions (Part 2 of 4)

Executable	Description
Assembler quartus_asm	The Quartus II Assembler generates a device programming image, in the form of one or more of the following from a successful fit (that is, place-and-route). ● Programmer Object Files (**.pof**) ● SRAM Object Files (**.sof**) ● Hexadecimal (Intel-Format) Output Files (**.hexout**) ● Tabular Text Files (**.ttf**) ● Raw Binary Files (**.rbf**) The **.pof** and **.sof** files are then processed by the Quartus II Programmer and downloaded to the device with the MasterBlaster™ or the ByteBlaster™ II download cable, or the Altera Programming Unit (APU). The Hexadecimal (Intel-Format) Output Files, Tabular Text Files, and Raw Binary Files can be used by other programming hardware manufacturers that provide support for Altera devices. The Quartus II Fitter must be run successfully before running the Assembler.
Classic Timing Analyzer quartus_tan	The Quartus II Classic Timing Analyzer computes delays for the given design and device, and annotates them on the netlist. Then, the Classic Timing Analyzer performs timing analysis, allowing you to analyze the performance of all logic in your design. The **quartus_tan** executable includes Tcl support. Quartus II Analysis and Synthesis or the Fitter must be run successfully before running the Classic Timing Analyzer.
TimeQuest Timing Analyzer quartus_sta	The Quartus II TimeQuest Timing Analyzer computes delays for the given design and device, and annotates them on the netlist. Then, the TimeQuest Timing Analyzer performs timing analysis, allowing you to analyze the performance of all logic in your design. The **quartus_sta** executable includes Tcl support and SDC support. Quartus II Analysis and Synthesis or the Fitter must be run successfully before running the TimeQuest Timing Analyzer.
Design Assistant quartus_drc	The Quartus II Design Assistant checks the reliability of a design based on a set of design rules. The Design Assistant is especially useful for checking the reliability of a design before converting the design for HardCopy® devices. The Design Assistant supports designs that target any Altera device supported by the Quartus II software, except MAX® 3000 and MAX 7000 devices. Quartus II Analysis and Synthesis or the Fitter must be run successfully before running the Design Assistant.
Compiler Database Interface quartus_cdb	The Quartus II Compiler Database Interface generates incremental netlists for use with LogicLock™ back-annotation, or back-annotates device and resource assignments to preserve the fit for future compilations. The **quartus_cdb** executable includes Tcl support. Analysis and Synthesis must be run successfully before running the Compiler Database Interface.

Table 2–1. Quartus II Command-Line Executables and Descriptions (Part 3 of 4)

Executable	Description
EDA Netlist Writer quartus_eda	The Quartus II EDA Netlist Writer generates netlist and other output files for use with other EDA tools. Analysis and Synthesis, the Fitter, or Timing Analyzer must be run successfully before running the EDA Netlist Writer, depending on the arguments used.
Simulator quartus_sim	The Quartus II Simulator tests and debugs the logical operation and internal timing of the design entities in a project. The Simulator can perform two types of simulation: functional simulation and timing simulation. The **quartus_sim** executable includes Tcl support. Quartus II Analysis and Synthesis must be run successfully before running a functional simulation. The Timing Analyzer must be run successfully before running a timing simulation.
Power Analyzer quartus_pow	The Quartus II PowerPlay Power Analyzer estimates the thermal dynamic power and the thermal static power consumed by the design. For newer families such as Stratix® II and MAX II, the power drawn from each power supply is also estimated. Quartus II Analysis and Synthesis or the Fitter must be run successfully before running the PowerPlay Power Analyzer.
Programmer quartus_pgm	The Quartus II Programmer programs Altera devices. The Programmer uses one of the supported file formats: • Programmer Object Files (**.pof**) • SRAM Object Files (**.sof**) • Jam File (**.jam**) • Jam Byte-Code File (**.jbc**) Make sure you specify a valid programming mode, programming cable, and operation for a specified device.
Convert Programming File quartus_cpf	The Quartus II Convert Programming File module converts one programing file format to a different possible format. Make sure you specify valid options and an input programming file to generate the new requested programming file format.
Quartus Shell quartus_sh	The Quartus II Shell acts as a simple Quartus II Tcl interpreter. The Shell has a smaller memory footprint than the other command-line executables that support Tcl. The Shell may be started as an interactive Tcl interpreter (shell), used to run a Tcl script, or used as a quick Tcl command evaluator, evaluating the remaining command-line arguments as one or more Tcl commands.

Table 2–1. Quartus II Command-Line Executables and Descriptions (Part 4 of 4)

Executable	Description
TimeQuest Timing Analyzer GUI quartus_staw	This executable opens the Quartus II TimeQuest Timing Analyzer GUI. This is helpful because you don't have to open the entire Quartus II GUI for certain operations.
Programmer GUI quartus_pgmw	This executable opens up the programmer—a GUI to the quartus_pgm executable. This is helpful because users don't have to open the entire Quartus II GUI for certain operations

Command-Line Scripting Help

Help on command-line executables is available through different methods. You can access help built in to the executables with command-line options. You can use the Quartus II Command-Line and Tcl API Help browser for an easy graphical view of the help information. Additionally, you can refer to the *Scripting Reference Manual* on the Quartus II literature page on Altera's website, which has the same information in PDF format.

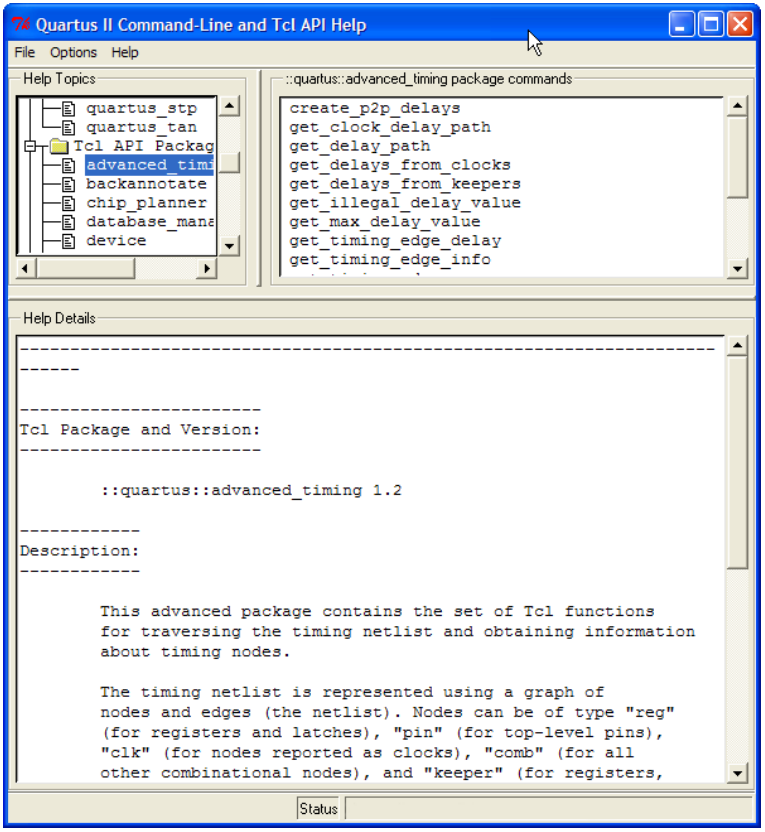
To use the Quartus II Command-Line and Tcl API Help browser, type the following:

```
quartus_sh --qhelp ↵
```

This command starts the Quartus II Command-Line and Tcl API Help browser, a viewer for information about the Quartus II Command-Line executables and Tcl API (Figure 2–1).

Use the `-h` option with any of the Quartus II Command-Line executables to get a description and list of supported options. Use the `--help=<`*option name*`>` option for detailed information about each option.

Figure 2–1. Quartus II Command-Line and Tcl API Help Browser



Command-Line Option Details

Command-line options are provided for many common global project settings and performing common tasks. You can use either of two methods to make assignments to an individual entity. If the project exists, open the project in the Quartus II GUI, change the assignment, and close the project. The changed assignment is updated in the Quartus II Settings File. Any command-line executables that are run after this update use the updated assignment. Refer to "Option Precedence" on page 2–8 for more

information. You can also make assignments using the Quartus II Tcl scripting API. If you want to completely script the creation of a Quartus II project, choose this method.



Refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Scripting information for all Quartus II project settings and assignments is located in the *QSF Reference Manual*.

Option Precedence

If you use command-line executables, you should be aware of the precedence of various project assignments and how to control the precedence. Assignments for a particular project exist in the Quartus II Settings File for the project. Assignments for a project can also be made with command-line options, as described earlier in this document. Project assignments are reflected in compiler database files that hold intermediate compilation results and reflect assignments made in the previous project compilation.

All command-line options override any conflicting assignments found in the Quartus II Settings File or the compiler database files. There are two command-line options to specify whether Quartus II Settings File or compiler database files take precedence for any assignments not specified as command-line options.



Any assignment not specified as a command-line option or found in the Quartus II Settings File or compiler database file is set to its default value.

The file precedence command-line options are `--read_settings_files` and `--write_settings_files`.

By default, the `--read_settings_files` and `--write_settings_files` options are turned on. Turning on the `--read_settings_files` option causes a command-line executable to read assignments from the Quartus II Settings File instead of from the compiler database files. Turning on the `--write_settings_files` option causes a command-line executable to update the Quartus II Settings File to reflect any specified options, as happens when closing a project in the Quartus II GUI.

Table 2–2 lists the precedence for reading assignments depending on the value of the `--read_settings_files` option.

Table 2–2. Precedence for Reading Assignments

Option Specified	Precedence for Reading Assignments
--read_settings_files = `on` (default)	1. Command-line options 2. Quartus II Settings File 3. Project database (**db** directory, if it exists) 4. Quartus II software defaults
--read_settings_files = `off`	1. Command-line options 2. Project database (**db** directory, if it exists) 3. Quartus II software defaults

Table 2–3 lists the locations to which assignments are written, depending on the value of the `--write_settings_files` command-line option.

Table 2–3. Location for Writing Assignments

Option Specified	Location for Writing Assignments
--write_settings_files = `on` (Default)	Quartus II Settings File and compiler database
--write_settings_files = `off`	Compiler database

Example 2–3 assumes that a project named **fir_filter** exists, and that the analysis and synthesis step has been performed (using the **quartus_map** executable).

Example 2–3. Write Settings Files

```
quartus_fit fir_filter --fmax=80MHz ↵
quartus_tan fir_filter ↵
quartus_tan fir_filter --fmax=100MHz --tao=timing_result-100.tao
    --write_settings_files=off ↵
```

The first command, **quartus_fit fir_filter --fmax=80MHz**, runs the **quartus_fit** executable and specifies a global f_{MAX} requirement of 80 MHz.

The second command, **quartus_tan fir_filter**, runs Quartus II timing analysis for the results of the previous fit.

The third command reruns Quartus II timing analysis with a global f_{MAX} requirement of 100 MHz and saves the result in a file called **timing_result-100.tao**. By specifying the `--write_settings_files=off` option, the command-line executable does not update the Quartus II Settings File to reflect the changed f_{MAX} requirement. The compiler database files reflect the changed f_{MAX} requirement. If the `--write_settings_files=off` option is not specified, the command-line executable updates the Quartus II Settings File to reflect the 100-MHz global f_{MAX} requirement.

Use the options `--read_settings_files=off` and `--write_settings_files=off` (where appropriate) to optimize the way that the Quartus II software reads and updates settings files. Example 2–4 shows how to avoid unnecessary reading and writing.

Example 2–4. Avoiding Unnecessary Reading and Writing

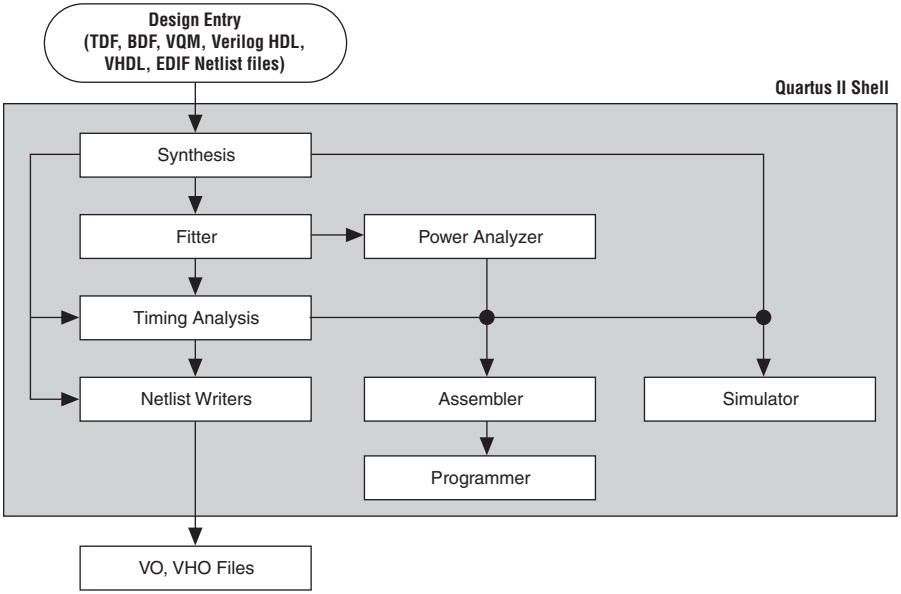
```
quartus_map filtref --source=filtref
    --part=ep1s10f780c5 ↵
quartus_fit filtref --fmax=100MHz
    --read_settings_files=off ↵
quartus_tan filtref --read_settings_files=off
    --write_settings_files=off ↵
quartus_asm filtref --read_settings_files=off
    --write_settings_files=off ↵
```

The **quartus_tan** and **quartus_asm** executables do not read or write settings files because they do not change any settings in the project.

Design Flow

Figure 2–2 shows a typical design flow.

Figure 2–2. Typical Design Flow



Compilation with quartus_sh --flow

Use the **quartus_sh** executable with the `--flow` option to perform a complete compilation flow with a single command. (For information about specialized flows, type `quartus_sh --help=flow` ↵ at a command prompt.) The `--flow` option supports the smart recompile feature and efficiently sets command-line arguments for each executable in the flow.

If you used the **quartus_cmd** executable to perform command-line compilations in earlier versions of the Quartus II software, you should use the `quartus_sh --flow` command beginning with the Quartus II software version 3.0.

The following example runs compilation, timing analysis, and programming file generation with a single command:

```
quartus_sh --flow compile filtref ↵
```

Text-Based Report Files

Each command-line executable creates a text report file when it is run. These files report success or failure, and contain information about the processing performed by the executable.

Report file names contain the revision name and the short-form name of the executable that generated the report file: <*revision*>.<*executable*>**.rpt**. For example, using the **quartus_fit** executable to place and route a project with the revision name **design_top** generates a report file named **design_top.fit.rpt**. Similarly, using the **quartus_tan** executable to perform timing analysis on a project with the revision name **fir_filter** generates a report file named **fir_filter.tan.rpt**.

As an alternative to parsing text-based report files, you can use the Tcl package called **::quartus::report**. For more information about this package, refer to "Command-Line Scripting Help" on page 2–6.

You can use Quartus II command-line executables in scripts that control a design flow that uses other software in addition to the Quartus II software. For example, if your design flow uses other synthesis or simulation software, and you can run the other software at a system command prompt, you can include it in a single script. The Quartus II command-line executables include options for common global project settings and operations, but you must use a Tcl script or the Quartus II GUI to set up a new project and apply individual constraints, such as pin location assignments and timing requirements. Command-line executables are very useful for working with existing projects, for making common global settings, and for performing common operations. For more flexibility in a flow, use a Tcl script, which makes it easier to pass data between different stages of the design flow and have more control during the flow.



For more information about Tcl scripts, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*, or the *Quartus II Scripting Reference Manual*.

For example, your script could run other synthesis software, then place-and-route the design in the Quartus II software, then generate output netlists for other simulation software. Example 2–5 shows how to do this with a UNIX shell script for a design that targets a Cyclone II device.

Example 2–5. Script for End-to-End Flow

```
#!/bin/sh
# Run synthesis first.
# This example assumes you use Synplify software
synplify -batch synthesize.tcl

# If your Quartus II project exists already, you can just
# recompile the design.
# You can also use the script described in a later example to
# create a new project from scratch
quartus_sh --flow compile myproject

# Use the quartus_tan executable to do best and worst case
# timing analysis
quartus_tan myproject --tao=worst_case
quartus_tan myproject --fast_model --tao=best_case

# Use the quartus_eda executable to write out a gate-level
# Verilog simulation netlist for ModelSim
quartus_eda my_project --simulation --tool=modelsim
 --format=verilog

# Perform the simulation with the ModelSim software
vlib cycloneii_ver
vlog -work cycloneii_ver c:/quartusii/eda/sim_lib/cycloneii_atoms.v
vlib work
vlog -work work my_project.vo
vsim -L cycloneii_ver -t 1ps work.my_project
```

Makefile Implementation

You can also use the Quartus II command-line executables in conjunction with the **make** utility to automatically update files when other files they depend on change. The file dependencies and commands used to update files are specified in a text file called a makefile.

To facilitate easier development of efficient makefiles, the following "smart action" scripting command is provided with the Quartus II software:

```
quartus_sh --determine_smart_action ↵
```

Because assignments for a Quartus II project are stored in the Quartus II Settings File (**.qsf**), including it in every rule results in unnecessary processing steps. For example, updating a setting related to programming file generation (which requires re-running only **quartus_asm**) modifies the Quartus II Settings File, requiring a complete recompilation if the Quartus II Settings File is included in every rule.

The smart action command determines the earliest command-line executable in the compilation flow that must be run based on the current Quartus II Settings File, and generates a change file corresponding to that executable. For a given command-line executable named **quartus_**<*executable*>, the change file is named with the format <*executable*>**.chg**. For example, if **quartus_map** must be re-run, the smart action command creates or updates a file named **map.chg**. Thus, rather than including the Quartus II Settings File in each makefile rule, include only the appropriate change file.

Example 2–6 uses change files and the smart action command. You can copy and modify it for your own use. A copy of this example is included in the help for the makefile option, which is available by typing:

```
quartus_sh --help=makefiles ↵
```

Example 2–6. Sample Makefile

```
###################################################################
# Project Configuration:
#
# Specify the name of the design (project), the Quartus II Settings
# File (.qsf), and the list of source files used.
###################################################################

PROJECT = chiptrip
SOURCE_FILES = auto_max.v chiptrip.v speed_ch.v tick_cnt.v time_cnt.v
ASSIGNMENT_FILES = chiptrip.qpf chiptrip.qsf

###################################################################
# Main Targets
#
# all: build everything
# clean: remove output files and database
###################################################################

all: smart.log $(PROJECT).asm.rpt $(PROJECT).tan.rpt

clean:
    rm -rf *.rpt *.chg smart.log *.htm *.eqn *.pin *.sof *.pof db

map: smart.log $(PROJECT).map.rpt
fit: smart.log $(PROJECT).fit.rpt
asm: smart.log $(PROJECT).asm.rpt
tan: smart.log $(PROJECT).tan.rpt
smart: smart.log

###################################################################
# Executable Configuration
###################################################################

MAP_ARGS = --family=Stratix
FIT_ARGS = --part=EP1S20F484C6
ASM_ARGS =
TAN_ARGS =

###################################################################
# Target implementations
###################################################################

STAMP = echo done >

$(PROJECT).map.rpt: map.chg $(SOURCE_FILES)
    quartus_map $(MAP_ARGS) $(PROJECT)
    $(STAMP) fit.chg

$(PROJECT).fit.rpt: fit.chg $(PROJECT).map.rpt
    quartus_fit $(FIT_ARGS) $(PROJECT)
```

```
    $(STAMP) asm.chg
    $(STAMP) tan.chg

$(PROJECT).asm.rpt: asm.chg $(PROJECT).fit.rpt
    quartus_asm $(ASM_ARGS) $(PROJECT)

$(PROJECT).tan.rpt: tan.chg $(PROJECT).fit.rpt
    quartus_tan $(TAN_ARGS) $(PROJECT)

smart.log: $(ASSIGNMENT_FILES)
    quartus_sh --determine_smart_action $(PROJECT) > smart.log

######################################################################
# Project initialization
######################################################################

$(ASSIGNMENT_FILES):
    quartus_sh --prepare $(PROJECT)

map.chg:
    $(STAMP) map.chg
fit.chg:
    $(STAMP) fit.chg
tan.chg:
    $(STAMP) tan.chg
asm.chg:
    $(STAMP) asm.chg
```

A Tcl script is provided with the Quartus II software to create or modify files that can be specified as dependencies in the make rules, assisting you in makefile development. Complete information about this Tcl script and how to integrate it with makefiles is available by running the following command:

```
quartus_sh --help=determine_smart_action ↵
```

Command-Line Scripting Examples

This section of the chapter presents various examples of command-line executable use.

Create a Project and Apply Constraints

The command-line executables include options for common global project settings and commands. To apply constraints such as pin locations and timing assignments, run a Tcl script with the constraints in it. You can write a Tcl constraint file from scratch, or generate one for an existing project. From the Project menu, click **Generate Tcl File for Project**.

Example 2–7 creates a project with a Tcl script and applies project constraints using the tutorial design files in the <*Quartus II installation directory*>**/qdesigns/fir_filter/** directory.

Example 2–7. Tcl Script to Create Project and Apply Constraints

```
project_new filtref -overwrite
# Assign family, device, and top-level file
set_global_assignment -name FAMILY Cyclone
set_global_assignment -name DEVICE EP1C12Q240C6
set_global_assignment -name BDF_FILE filtref.bdf
# Assign pins
set_location_assignment -to clk Pin_28
set_location_assignment -to clkx2 Pin_29
set_location_assignment -to d[0] Pin_139
set_location_assignment -to d[1] Pin_140
# Other pin assignments could follow
# Create timing assignments
create_base_clock -fmax "100 MHz" -target clk clocka
create_relative_clock -base_clock clocka -divide 2 \
    -offset "500 ps" -target clkx2 clockb
set_multicycle_assignment -from clk -to clkx2 2
# Other timing assignments could follow
project_close
```

Save the script in a file called **setup_proj.tcl** and type the commands illustrated in Example 2–8 at a command prompt to create the design, apply constraints, compile the design, and perform fast-corner and slow-corner timing analysis. Timing analysis results are saved in two files.

Example 2–8. Script to Create and Compile a Project

```
quartus_sh -t setup_proj.tcl ↵
quartus_map filtref ↵
quartus_fit filtref ↵
quartus_asm filtref ↵
quartus_tan filtref --fast_model --tao=min.tao
    --export_settings=off ↵
    quartus_tan filtref --tao=max.tao
    --export_settings=off ↵
```

You can use the following two commands to create the design, apply constraints, and compile the design:

```
quartus_sh -t setup_proj.tcl ↵
quartus_sh --flow compile filtref ↵
```

The **quartus_sh --flow compile** command performs a full compilation, and is equivalent to clicking the **Start Compilation** button in the toolbar.

Check Design File Syntax

The UNIX shell script example shown in Example 2–9 assumes that the Quartus II software **fir_filter** tutorial project exists in the current directory. (You can find the **fir_filter** project in the <*Quartus II directory*>**/qdesigns/fir_filter** directory unless the Quartus II software tutorial files are not installed.)

The `--analyze_file` option performs a syntax check on each file. The script checks the exit code of the **quartus_map** executable to determine whether there is an error during the syntax check. Files with syntax errors are added to the `FILES_WITH_ERRORS` variable, and when all files are checked, the script prints a message indicating syntax errors. When options are not specified, the executable uses the project database values. If not specified in the project database, the executable uses the Quartus II software default values. For example, the **fir_filter** project is set to target the Cyclone device family, so it is not necessary to specify the `--family` option.

Example 2–9. Shell Script to Check Design File Syntax

```
#!/bin/sh
FILES_WITH_ERRORS=""
# Iterate over each file with a .bdf or .v extension
for filename in `ls *.bdf *.v`
do
  # Perform a syntax check on the specified file
    quartus_map fir_filter --analyze_file=$filename
  # If the exit code is non-zero, the file has a syntax error
    if [ $? -ne 0 ]
    then
       FILES_WITH_ERRORS="$FILES_WITH_ERRORS $filename"
    fi
done
if [ -z "$FILES_WITH_ERRORS" ]
then
    echo "All files passed the syntax check"
    exit 0
else
    echo "There were syntax errors in the following file(s)"
    echo $FILES_WITH_ERRORS
    exit 1
fi
```

Create a Project and Synthesize a Netlist Using Netlist Optimizations

This example creates a new Quartus II project with a file **top.edf** as the top-level entity. The `--enable_register_retiming=on` and `--enable_wysiwyg_resynthesis=on` options allow the technology mapper to optimize the design using gate-level register retiming and technology remapping.



For more details about register retiming, WYSIWYG primitive resynthesis, and other netlist optimization options, refer to the Quartus II Help.

The `--part` option tells the technology mapper to target an EP20K600EBC652-1X device. To create the project and synthesize it using the netlist optimizations described above, type the command shown in Example 2–10 at a command prompt.

Example 2–10. Creating a Project and Synthesizing a Netlist Using Netlist Optimizations

```
quartus_map top --source=top.edf --enable_register_retiming=on
    --enable_wysiwyg_resynthesis=on --part=EP20K600EBC652-1X ↵
```

Archive and Restore Projects

You can archive or restore a Quartus II project with a single command. This makes it easy to take snapshots of projects when you use batch files or shell scripts for compilation and project management. Use the `--archive` or `--restore` options for **quartus_sh** as appropriate. Type the command shown in Example 2–11 at a system command prompt to archive your project.

Example 2–11. Archiving a Project

```
quartus_sh --archive <project name> ↵
```

The archive file is automatically named <*project name*>.**qar**. If you want to use a different name, rename the archive after it has been created. This command overwrites any existing archive with the same name.

To restore a project archive, type the command shown in Example 2–12 at a system command prompt:

Example 2–12. Restoring a Project Archive

```
quartus_sh --restore <archive name> ↵
```

The command restores the project archive to the current directory and overwrites existing files.

Perform I/O Assignment Analysis

You can perform I/O assignment analysis with a single command. I/O assignment analysis checks pin assignments to ensure they do not violate board layout guidelines. I/O assignment analysis does not require a complete place and route, so it is a quick way to ensure your pin assignments are correct. The command shown in Example 2–13 performs I/O assignment analysis for the specified project and revision.

Example 2–13. Performing I/O Assignment Analysis

```
quartus_fit --check_ios <project name> --rev=<revision name> ↵
```

Update Memory Contents without Recompiling

You can use two simple commands to update the contents of memory blocks in your design without recompiling. Use the **quartus_cdb** executable with the `--update_mif` option to update memory contents from Memory Initialization Files or Hexadecimal (Intel-Format) Files. Then re-run the assembler with the **quartus_asm** executable to regenerate the SOF, POF, and any other programming files.

Example 2–14 shows these two commands.

Example 2–14. Commands to Update Memory Contents without Recompiling

```
quartus_cdb --update_mif <project name> [--rev=<revision name>] ↵
quartus_asm <project name> [--rev=<revision name>] ↵
```

Example 2–15 shows the commands for a DOS batch file for this example. You can paste the following lines into a DOS batch file called **update_memory.bat**.

Example 2–15. Batch file to Update Memory Contents without Recompiling

```
quartus_cdb --update_mif %1 --rev=%2
quartus_asm %1 --rev=%2
```

Type the following command at a system command prompt:

```
update_memory.bat <project name> <revision name> ↵
```

Fit a Design as Quickly as Possible

This example assumes that a project called **top** exists in the current directory, and that the name of the top-level entity is **top**. The `--effort=fast` option causes the Fitter to use the fast fit algorithm to increase compilation speed, possibly at the expense of reduced f_{MAX} performance. The `--one_fit_attempt=on` option restricts the Fitter to only one fitting attempt for the design.

To attempt to fit the project called **top** as quickly as possible, type command shown in Example 2–16 at a command prompt.

Example 2–16. Fitting a Project Quickly

```
quartus_fit top --effort=fast --one_fit_attempt=on ↵
```

Fit a Design Using Multiple Seeds

This shell script example assumes that the Quartus II software tutorial project called **fir_filter** exists in the current directory (defined in the file **fir_filter.qpf**). If the tutorial files are installed on your system, this project exists in the <*Quartus II directory*>**/qdesigns**<*quartus_version_number*>**/fir_filter** directory. Because the top-level entity in the project does not have the same name as the project, you must specify the revision name for the top-level entity with the `--rev` option. The `--seed` option specifies the seeds to use for fitting.

A seed is a parameter that affects the random initial placement of the Quartus II Fitter. Varying the seed can result in better performance for some designs.

After each fitting attempt, the script creates new directories for the results of each fitting attempt and copies the complete project to the new directory so that the results are available for viewing and debugging after the script has completed.

Example 2–17 is designed for use on UNIX systems using **sh** (the shell).

Example 2–17. Shell Script to Fit a Design Using Multiple Seeds

```
#!/bin/sh
ERROR_SEEDS=""
quartus_map fir_filter --rev=filtref
# Iterate over a number of seeds
for seed in 1 2 3 4 5
do
echo "Starting fit with seed=$seed"
# Perform a fitting attempt with the specified seed
        quartus_fit fir_filter --seed=$seed --rev=filtref
# If the exit-code is non-zero, the fitting attempt was
# successful, so copy the project to a new directory
        if [ $? -eq 0 ]
        then
          mkdir ../fir_filter-seed_$seed
          mkdir ../fir_filter-seed_$seed/db
          cp * ../fir_filter-seed_$seed
          cp db/* ../fir_filter-seed_$seed/db
        else
          ERROR_SEEDS="$ERROR_SEEDS $seed"
        fi
done
if [ -z "$ERROR_SEEDS" ]
then
        echo "Seed sweeping was successful"
        exit 0
else
        echo "There were errors with the following seed(s)"
        echo $ERROR_SEEDS
        exit 1
fi
```



Use the Design Space Explorer included with the Quartus II software (DSE) script (by typing `quartus_sh --dse` ↵ at a command prompt) to improve design performance by performing automated seed sweeping.



For more information about the DSE, type `quartus_sh --help=dse` ↵ at the command prompt, or refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*, or see the Quartus II Help.

The QFlow Script

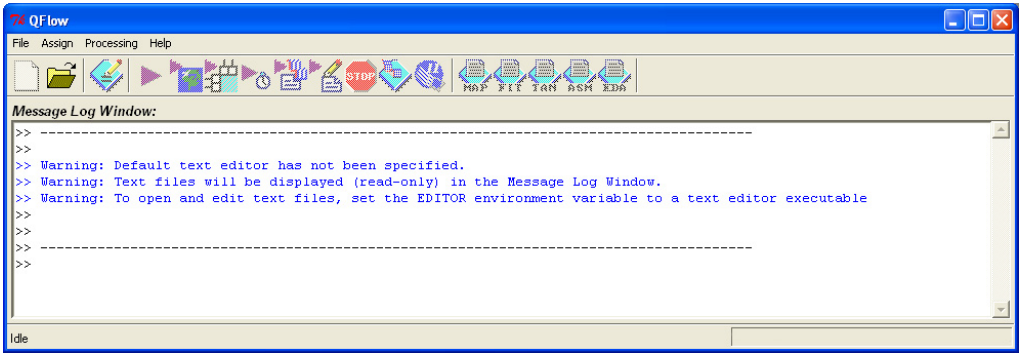
A Tcl/Tk-based graphical interface called QFlow is included with the command-line executables. You can use the QFlow interface to open projects, launch some of the command-line executables, view report files, and make some global project assignments. The QFlow interface can run the following command-line executables:

- **quartus_map** (Analysis and Synthesis)
- **quartus_fit** (Fitter)
- **quartus_tan** (Timing Analysis)
- **quartus_asm** (Assembler)
- **quartus_eda** (EDA Netlist Writer)

To view floorplans or perform other GUI-intensive tasks, launch the Quartus II software.

Start QFlow by typing the following command at a command prompt: `quartus_sh -g` ↵ . Figure 2–3 shows the QFlow interface.

Figure 2–3. QFlow Interface





The QFlow script is located in the <*Quartus II directory*>**/common/tcl/apps/qflow/** directory.

Referenced Documents

This chapter references the following documents:

- *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 2–4 shows the revision history for this document.

Table 2–4. Document Revision History

Date / Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Added "Referenced Documents" on page 2–23.	Updated for the Quartus II software version 7.2.
May 2007 v7.1.0	Updated Quartus II software 7.1 revision, including making very minor updates to: • "Introductory Example" on page 2–2 • Part of the running text on page 2–17 • Updated screenshot for Figure 2–1 on page 2–7	Updated Quartus II software 7.1 release.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Added revision history for this document.	—
May 2006 v6.0.0	Added the Quartus II TimeQuest Timing Analyzer feature.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.0.	—
May 2005 v5.0.0	Updated for the Quartus II software version 5.0.0.	—
Dec. 2004 v2.1	• Updates to tables and figures. • New functionality in the Quartus II software version 4.2.	—
June 2004 v2.0	• Updates to tables and figures. • New functionality in the Quartus II software version 4.1.	—
Feb. 2004 v1.0	Initial release.	—

3. Tcl Scripting

QII52003-7.2.0

Introduction

Developing and running tool command language (Tcl) scripts to control the Altera® Quartus® II software allows you to perform a wide range of functions, such as compiling a design or writing procedures to automate common tasks.

You can use Tcl scripts to manage a Quartus II project, make assignments, define design constraints, make device assignments, run compilations, perform timing analysis, import LogicLock™ region assignments, use the Quartus II Chip Editor, and access reports. You can automate your Quartus II assignments using Tcl scripts so that you do not have to create them individually. Tcl scripts also facilitate project or assignment migration. For example, when using the same prototype or development board for different projects, you can automate reassignment of pin locations in each new project. The Quartus II software can also generate a Tcl script based on all the current assignments in the project, which aids in switching assignments to another project.

The Quartus II software Tcl commands follow the EDA industry Tcl application programming interface (API) standards for using command-line options to specify arguments. This simplifies learning and using Tcl commands. If you encounter an error using a command argument, the Tcl interpreter gives help information showing correct usage.

This chapter includes sample Tcl scripts for automating the Quartus II software. You can modify these example scripts for use with your own designs. You can find more Tcl scripts in the Design Examples section of the Support area of Altera's website.

This chapter includes the following topics:

- "Introduction"
- "Quartus II Tcl Packages" on page 3–2
- "Quartus II Tcl API Help" on page 3–5
- "Executables Supporting Tcl" on page 3–9
- "End-to-End Design Flows" on page 3–12
- "Creating Projects and Making Assignments" on page 3–13
- "Compiling Designs" on page 3–21
- "Reporting" on page 3–22
- "Timing Analysis" on page 3–25
- "Automating Script Execution" on page 3–30

- "Other Scripting Features" on page 3–33
- "Using the Quartus II Tcl Shell in Interactive Mode" on page 3–39
- "Quartus II Legacy Tcl Support" on page 3–42
- "Tcl Scripting Basics" on page 3–42

What is Tcl?

Tcl (pronounced "tickle") is a popular scripting language that is similar to many shell scripting and high-level programming languages. It provides support for control structures, variables, network socket access, and APIs. Tcl is the EDA industry-standard scripting language used by Synopsys, Mentor Graphics®, Synplicity, and Altera software. It allows you to create custom commands and works seamlessly across most development platforms. For a list of recommended literature on Tcl, refer to "External References" on page 3–50.

You can create your own procedures by writing scripts containing basic Tcl commands, user-defined procedures, and Quartus II API functions. You can then automate your design flow, run the Quartus II software in batch mode, or execute the individual Tcl commands interactively in the Quartus II Tcl interactive shell.

If you're unfamiliar with Tcl scripting, or are a Tcl beginner, refer to the "Tcl Scripting Basics" on page 3–42 for an introduction to Tcl scripting.

The Quartus II software, beginning with version 4.1, supports Tcl/Tk version 8.4, supplied by the Tcl DeveloperXchange at **tcl.activestate.com**.

Quartus II Tcl Packages

The Quartus II Tcl commands are grouped in packages by function. Table 3–1 describes each Tcl package.

Table 3–1. Tcl Packages (Part 1 of 2)

Package Name	Package Description
advanced_timing	Traverse the timing netlist and get information about timing nodes
backannotate	Back annotate assignments
chip_planner	Identify and modify resource usage and routing with the Chip Editor
database_manager	Manage version-compatible database files
device	Get device and family information from the device database
flow	Compile a project, run command-line executables and other common flows
insystem_memory_edit	Read and edit memory contents in Altera devices
jtag	Control the jtag chain

Table 3–1. Tcl Packages (Part 2 of 2)

Package Name	Package Description
logic_analyzer_interface	Query and modify the logic analyzer interface output pin state
logiclock	Create and manage LogicLock regions
misc	Perform miscellaneous tasks
project	Create and manage projects and revisions, make any project assignments including timing assignments
report	Get information from report tables, create custom reports
sdc	Specifies constraints and exceptions to the TimeQuest Analyzer
simulator	Configure and perform simulations
sta	Contains the set of Tcl functions for obtaining advanced information from the Quartus II TimeQuest Timing Analyzer
stp	Run the SignalTap® II logic analyzer
timing	Annotate timing netlist with delay information, compute and report timing paths
timing_assignment	Contains the set of Tcl functions for making project-wide timing assignments, including clock assignments; all Tcl commands designed to process Quartus II Classic Timing Analyzer assignments have been moved to this package
timing_report	List timing paths
sdc_ext	Altera-specific SDC commands

By default, only the minimum number of packages is loaded automatically with each Quartus II executable. This keeps the memory requirement for each executable as low as possible. Because the minimum number of packages is automatically loaded, you must load other packages before you can run commands in those packages.

Table 3–2 lists the Quartus II Tcl packages available with Quartus II executables and indicates whether a package is loaded by default (●) or is available to be loaded as necessary (◓). A clear circle (○) means that the package is not available in that executable.

Table 3–2. Tcl Package Availability by Quartus II Executable (Part 1 of 2)

Packages	Quartus II Executable						
	Quartus_sh	Quartus_tan	Quartus_cdb	Quartus_sim	Quartus_stp	Quartus_sta Quartus_staw	Tcl Console
advanced_timing	○	◓	○	○	○	○	○
backannotate	○	○	◓	○	○	○	◓

Table 3–2. Tcl Package Availability by Quartus II Executable (Part 2 of 2)

Packages	Quartus II Executable						
	Quartus_sh	Quartus_tan	Quartus_cdb	Quartus_sim	Quartus_stp	Quartus_sta Quartus_staw	Tcl Console
chip_planner	○	○	◓	○	○	○	○
device	●	◓	●	●	○	●	◓
flow	◓	◓	◓	◓	○	◓	◓
insystem_memory_edit	○	○	○	○	●	○	○
jtag	○	○	○	○	●	○	○
logic_analyzer_ interface	○	○	○	○	●	○	○
logiclock	○	◓	◓	○	○	○	◓
misc	●	●	●	●	●	●	●
old_api	○	○	○	○	○	○	●
project	●	●	●	●	●	●	●
report	◓	◓	◓	●	○	●	◓
sdc	○	○	○	○	○	●	○
sdc_ext	○	○	○	○	○	●	○
simulator	○	○	○	●	○	○	○
sta	○	○	○	○	○	●	○
stp	○	○	○	○	●	○	○
timing	○	●	○	○	○	○	○
timing_assignment	●	●	●	●	●	○	○
timing_report	○	◓	○	○	●	○	●

Notes to Table 3–2:

(1) A dark circle (●) indicates that the package is loaded automatically.

(2) A half-circle (◓) means that the package is available but not loaded automatically.

(3) A white circle (○) means that the package is not available for that executable.

Because different packages are available in different executables, you must run your scripts with executables that include the packages you use in the scripts. For example, if you use commands in the **timing** package, you must use the **quartus_tan** executable to run the script because the **quartus_tan** executable is the only one with support for the **timing** package.

Loading Packages

To load a Quartus II Tcl package, use the **load_package** command as follows:

```
load_package [-version <version number>] <package name>
```

This command is similar to the **package require** Tcl command (described in Table 3–3 on page 3–6), but you can easily alternate between different versions of a Quartus II Tcl package with the **load_package** command.



For additional information about these and other Quartus II command-line executables, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Quartus II Tcl API Help

Access the Quartus II Tcl API Help reference by typing the following command at a system command prompt:

```
quartus_sh --qhelp ↵
```

This command runs the Quartus II Command-Line and Tcl API help browser, which documents all commands and options in the Quartus II Tcl API. It includes detailed descriptions and examples for each command.

In addition, the information in the Tcl API help is available in the *Quartus II Scripting Reference Manual*, which is available in PDF on the Quartus II Literature page on the Altera website.

Quartus II Tcl help allows easy access to information about the Quartus II Tcl commands. To access the help information, type `help` at a Tcl prompt, as shown in Example 3–1.

Example 3–1. Help Output

```
tcl> help
--------------------------------------------------------------------------------

----------------------------------
Available Quartus II Tcl Packages:
----------------------------------

Loaded                          Not Loaded
-----------------------------   ----------------------
::quartus::misc                 ::quartus::device
::quartus::old_api              ::quartus::backannotate
::quartus::project              ::quartus::flow
::quartus::timing_assignment    ::quartus::logiclock
::quartus::timing_report        ::quartus::report


* Type "help -tcl"
  to get an overview on Quartus II Tcl usages.
```

Using the `-tcl` option with `help` displays an introduction to the Quartus II Tcl API that focuses on how to get help for Tcl commands (short help and long help) and Tcl packages.



The Tcl API help is also available in Quartus II online help. Search for the command or package name to find details about that command or package.

Table 3–3 summarizes the help options available in the Tcl environment.

Table 3–3. Help Options Available in the Quartus II Tcl Environment (Part 1 of 3)

Help Command	Description
`help`	To view a list of available Quartus II Tcl packages, loaded and not loaded.
`help -tcl`	To view a list of commands used to load Tcl packages and access command-line help.

Table 3–3. Help Options Available in the Quartus II Tcl Environment (Part 2 of 3)

Help Command	Description
`help -pkg` <*package_name*> `[-version` <*version number*>`]`	To view help for a specified Quartus II package that includes the list of available Tcl commands. For convenience, you can omit the `::quartus::` package prefix, and type `help -pkg` <*package name*> ↵. If you do not specify the `-version` option, help for the currently loaded package is displayed by default. If the package for which you want help is not loaded, help for the latest version of the package is displayed by default. Examples: `help -pkg ::quartus::p` ↵ `help -pkg ::quartus::project` ↵ `help -pkg project rhelp -pkg project -version 1.0` ↵
<*command_name*> `-h` or <*command_name*> `-help`	To view short help for a Quartus II Tcl command for which the package is loaded. Examples: `project_open -h` ↵ `project_open -help` ↵
`package require ::quartus::`<*package name*> `[`<*version*>`]`	To load a Quartus II Tcl package with the specified version. If <*version*> is not specified, the latest version of the package is loaded by default. Example: `package require ::quartus::project 1.0` ↵ This command is similar to the **load_package** command. The advantage of using **load_package** is that you can alternate freely between different versions of the same package. Type <*package name*> `[-version` <*version number*>`]` ↵ to load a Quartus II Tcl package with the specified version. If the `-version` option is not specified, the latest version of the package is loaded by default. Example: `load_package ::quartus::project -version 1.0` ↵

Table 3–3. Help Options Available in the Quartus II Tcl Environment (Part 3 of 3)

Help Command	Description
`help -cmd` <*command name*> `[-version` <*version number*>`]` *or* <*command name*> `-long_help`	To view long help for a Quartus II Tcl command. Only <*command name*> `-long_help` requires that the associated Tcl package is loaded. If you do not specify the `-version` option, help for the currently loaded package is displayed by default. If the package for which you want help is not loaded, help for the latest version of the package is displayed by default. Examples: `project_open -long_help` ↵ `help -cmd project_open` ↵ `help -cmd project_open -version 1.0` ↵
`help -examples`	To view examples of Quartus II Tcl usage.
`help -quartus`	To view help on the predefined global Tcl array that can be accessed to view information about the Quartus II executable that is currently running.
`quartus_sh --qhelp`	To launch the Tk viewer for Quartus II command-line help and display help for the command-line executables and Tcl API packages. For more information about this utility, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Executables Supporting Tcl

Some of the Quartus II command-line executables support Tcl scripting (refer to Table 3–4). Each executable supports different sets of Tcl packages. Refer to Table 3–4 to determine the appropriate executable to run your script.

Table 3–4. Command-line Executables Supporting Tcl Scripting

Executable Name	Executable Description
quartus_sh	The Quartus II Shell is a simple Tcl scripting shell, useful for making assignments, general reporting, and compiling.
quartus_tan	Use the Quartus II Classic Timing Analyzer to perform simple timing reporting and advanced timing analysis.
quartus_cdb	The Quartus II Compiler Database supports back annotation, LogicLock region operations, and Chip Editor functions.
quartus_sim	The Quartus II Simulator supports the automation of design simulation.
quartus_sta **quartus_staw**	The Quartus II TimeQuest Timing Analyzer supports SDC terminology for constraint entry and reporting.
quartus_stp	The Quartus II SignalTap II executable supports in-system debugging tools.

The **quartus_tan** and **quartus_cdb** executables support supersets of the packages supported by the **quartus_sh** executable. Use the **quartus_sh** executable if you run Tcl scripts with only project management and assignment commands, or if you need a Quartus II command-line executable with a small memory footprint.



For more information about these command-line executables, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Command-Line Options: -t, -s, and --tcl_eval

Table 3–5 lists three command-line options you can use with executables that support Tcl.

Table 3–5. Command-Line Options Supporting Tcl Scripting

Command-Line Option	Description
`-t <script file> [<script args>]`	Run the specified Tcl script with optional arguments.
`-s`	Open the executable in the interactive Tcl shell mode.
`--tcl_eval <tcl command>`	Evaluate the remaining command-line arguments as Tcl commands. For example, the following command displays help for the project package: `quartus_sh --tcl_eval help -pkg project`

Run a Tcl Script

Running an executable with the `-t` option runs the specified Tcl script. You can also specify arguments to the script. Access the arguments through the `argv` variable, or use a package such as **cmdline**, which supports arguments of the following form:

-<argument name> <argument value>

The **cmdline** package is included in the *<Quartus II directory>***/common/tcl/packages** directory.

For example, to run a script called **myscript.tcl** with one argument, `Stratix`, type the following command at a system command prompt:

```
quartus_sh -t myscript.tcl Stratix ↵
```



Beginning with version 4.1, the Quartus II software supports the `argv` variable. In previous software versions, script arguments are accessed in the `quartus(args)` global variable.

Refer to "Accessing Command-Line Arguments" on page 3–36 for more information.

Interactive Shell Mode

Running an executable with the `-s` option starts an interactive Tcl shell that displays a `tcl>` prompt. For example, type `quartus_tan -s` ↵ at a system command prompt to open the Quartus II Classic Timing

Analyzer executable in interactive shell mode. Commands you type in the Tcl shell are interpreted when you press **Enter**. You can run a Tcl script in the interactive shell with the following command:

```
source <script name> ↵
```

If a command is not recognized by the shell, it is assumed to be an external command and executed with the **exec** command.

Evaluate as Tcl

Running an executable with the `--tcl_eval` option causes the executable to immediately evaluate the remaining command-line arguments as Tcl commands. This can be useful if you want to run simple Tcl commands from other scripting languages.

For example, the following command runs the Tcl command that prints out the commands available in the **project** package.

```
quartus_sh --tcl_eval help -pkg project ↵
```

Using the Quartus II Tcl Console Window

You can run Tcl commands directly in the Quartus II Tcl Console window. On the View menu, click **Utility Windows**. By default, the Tcl Console window is docked in the bottom-right corner of the Quartus II GUI. Everything typed in the Tcl Console is interpreted by the Quartus II Tcl shell.



The Quartus II Tcl Console window supports the Tcl API used in the Quartus II software version 3.0 and earlier for backward compatibility with older designs and EDA tools.

Tcl messages appear in the **System** tab (Messages window). Errors and messages written to `stdout` and `stderr` also are shown in the Quartus II Tcl Console window.

Note that you can do limited timing analysis in the Tcl console in the Quartus II GUI. With the **timing_report** package, you can use the **list_path** command to get details on paths listed in the timing report. However, if you want to get information about timing paths that are not listed in the timing report, you must use the **quartus_tan** executable in shell mode or run a script that reports on the paths in which you are interested.

If your design uses the Quartus II TimeQuest Timing Analyzer, you should perform scripted timing analysis in the TimeQuest Tcl console.

As Table 3–2 shows, the Tcl console in the Quartus II GUI does not include support for every package, so you cannot run scripts that use commands in packages that are not supported.

End-to-End Design Flows

You can use Tcl scripts to control all aspects of the design flow, including controlling other software if it includes a scripting interface.

Typically, EDA tools include their own script interpreters that extend core language functionality with tool-specific commands. For example, the Quartus II Tcl interpreter supports all core Tcl commands, and adds numerous commands specific to the Quartus II software. You can include commands in one Tcl script to run another script, which allows you to combine or chain together scripts to control different tools. Because scripts for different tools must be executed with different Tcl interpreters, it is difficult to pass information between the scripts unless one script writes information into a file and another script reads it.

Within the Quartus II software, you can perform many different operations in a design flow (such as synthesis, fitting, and timing analysis) from a single script, making it easy to maintain global state information and pass data between the operations. However, there are some limitations on the operations you can perform in a single script due to the various packages supported by each executable. For example, you cannot write a single script that performs simulation with commands in the **simulator** package while using commands in the **advanced_timing** package; those two packages are not available in the same executable. In a case where you wanted to include Tcl simulation and advanced timing analysis commands, you must write two scripts.

There are no limitations on running flows from any executable. Flows include operations found in the Start section of the Processing menu in the Quartus II GUI, and are also documented with the **execute_flow** Tcl command. If you can make settings in the Quartus II software and run a flow to get your desired result, you can make the same settings and run the same flow in any command-line executable.

To revisit the example with simulation and timing analysis, you could write one script that includes settings that configure a simulation, with settings that configure timing analysis. Then, run the simulation and timing analysis flows with the **execute_flow** command.

Configuring a simulation includes specifying settings such as name and location of the stimulus file, the duration of the simulation, whether to perform glitch detection or not, and more. Configuring timing analysis includes specifying settings such as the required clock frequency, the number of paths to report, and which timing model to use. You can make the settings, then run the flows with the **execute_flow** command, in any Quartus II command-line executable.

Creating Projects and Making Assignments

One benefit of the Tcl scripting API is that it is easy to create a script that makes all the assignments for an existing project. You can use the script at any time to restore your project settings to a known state. From the Project menu, click **Generate Tcl File for Project** to automatically generate a Tcl file with all of your assignments. You can source this file to recreate your project, and you can edit the file to add other commands, such as compiling the design. The file is a good starting point to learn about project management commands and assignment commands.



Refer to "Interactive Shell Mode" on page 3–10 for information about sourcing a script. Scripting information for all Quartus II project settings and assignments is located in the *QSF Reference Manual*.

Example 3–2 shows how to create a project, make assignments, and compile the project. It uses the **fir_filter** tutorial design files.

Example 3–2. Create and Compile a Project

```
load_package flow

# Create the project and overwrite any settings
# files that exist
project_new fir_filter -revision filtref -overwrite
# Set the device, the name of the top-level BDF,
# and the name of the top level entity
set_global_assignment -name FAMILY Cyclone
set_global_assignment -name DEVICE EP1C6F256C6
set_global_assignment -name BDF_FILE filtref.bdf
set_global_assignment -name TOP_LEVEL_ENTITY filtref
# Add other pin assignments here
set_location_assignment -to clk Pin_G1
# Create a base clock and a derived clock
create_base_clock -fmax "100 MHz" -target clk clocka
create_relative_clock -base_clock clocka -divide 2 \
    -offset "500 ps" -target clkx2 clockb
# Create a multicycle assignment of 2 between
# the two clock domains in the design.
set_multicycle_assignment -from clk -to clkx2 2
execute_flow -compile
project_close
```



The assignments created or modified while a project is open are not committed to the Quartus II settings files unless you explicitly call **export_assignments** or **project_close** (unless `-dont_export_assignments` is specified). In some cases, such as when running **execute_flow**, the Quartus II software automatically commits the changes.

HardCopy Device Design



For information about using a scripted design flow for HardCopy II designs, refer to the *Script-Based Design for HardCopy Devices* chapter of the *HardCopy Handbook*. It contains sample scripts and recommendations to make your HardCopy II design flow easy.

A separate chapter in the *HardCopy Handbook* called *Timing Constraints for HardCopy II Devices* also contains information about script-based design for HardCopy II devices, with an emphasis on timing constraints.

EDA Tool Assignments

You can target EDA tools for a project in the Quartus II software in Tcl with the **set_global_assignment** Tcl command. To use the default tool settings for each EDA tool, you need only specify the EDA tool to be used. The EDA interfaces available for the Quartus II software cover design

entry, simulation, timing analysis, and board design tools. More advanced EDA tools such as those for formal verification and resynthesis are supported by their own global assignment.

By default, the EDA interface options are set to <none>. Table 3–6 lists the EDA interface options available in the Quartus II software. Enclose interface assignment options that contain spaces in quotation marks.

Table 3–6. EDA Interface Options in the Quartus II Software (Part 1 of 2)

Option	Acceptable Values
Design Entry (EDA_DESIGN_ENTRY_SYNTHESIS_TOOL)	• Design Architect • Design Compiler • FPGA Compiler • FPGA Compiler II • FPGA Compiler II Altera Edition • FPGA Express • LeonardoSpectrum™ • LeonardoSpectrum-Altera (Level 1) • Synplify • Synplify Pro • ViewDraw • Precision Synthesis • Custom
Simulation (EDA_SIMULATION_TOOL)	• ModelSim (VHDL output from the Quartus II software) • ModelSim (Verilog HDL output from the Quartus II software) • ModelSim-Altera (VHDL output from the Quartus II software) • ModelSim-Altera (Verilog HDL output from the Quartus II software) • SpeedWave • VCS • Verilog-XL • VSS • NC-Verilog (Verilog HDL output from the Quartus II software) • NC-VHDL (VHDL output from the Quartus II software) • Scirocco (VHDL output from the Quartus II software) • Custom Verilog HDL • Custom VHDL
Timing Analysis (EDA_TIMING_ANALYSIS_TOOL)	• PrimeTime (VHDL output from the Quartus II software) • PrimeTime (Verilog HDL output from the Quartus II software) • Stamp (board model) • Custom Verilog HDL • Custom VHDL

Table 3–6. EDA Interface Options in the Quartus II Software (Part 2 of 2)

Option	Acceptable Values
Board level tools (EDA_BOARD_DESIGN_TOOL)	• Signal Integrity (IBIS) • Symbol Generation (ViewDraw)
Formal Verification (EDA_FORMAL_VERIFICATION_TOOL)	• Conformal LEC
Resynthesis (EDA_RESYNTHESIS_TOOL)	• PALACE • Amplify

For example, to generate an NC-Sim Verilog simulation output file, EDA_SIMULATION_TOOL should be set to target NC-Sim Verilog as the desired output, as shown in Example 3–3.

Example 3–3.

```
set_global_assignment -name eda_simulation_tool \
"NcSim (Verilog HDL output from Quartus II)"
```



For information about using third-party simulation tools, refer to volume 3 of the *Quartus II Handbook*.

Example 3–4 shows compilation of the fir_filter design files, generating a VHDL Output (**.vho**) file output for NC-Sim Verilog simulation.

Example 3–4. Simple Design with .vho Output

```
# This script works with the quartus_sh executable
# Set the project name to filtref
set project_name filtref

# Open the Project. If it does not already exist, create it
if [catch {project_open $project_name}] {project_new \ $project_name}

# Set Family
set_global_assignment -name family APEX 20KE

# Set Device
set_global_assignment -name device ep20k100eqc208-1

# Optimize for speed
set_global_assignment -name optimization_technique speed

# Turn-on Fastfit fitter option to reduce compile times
set_global_assignment -name fast_fit_compilation on

# Generate a NC-Sim Verilog simulation Netlist
set_global_assignment -name eda_simulation_tool "NcSim\
(Verilog HDL output from Quartus II)"

# Create an FMAX=50MHz assignment called clk1 to pin clk
create_base_clock -fmax 50MHz -target clk clk1

# Create a pin assignment on pin clk
set_location -to clk Pin_134

# Compilation option 1
# Always write the assignments to the constraint files before
# doing a system call. Else, stand-alone files will not pick up
# the assignments
#export_assignments
#qexec quartus_map <project_name>
#qexec quartus_fit <project_name>
#qexec quartus_asm <project_name>
#qexec quartus_tan <project_name>
#qexec quartus_eda <project_name>

# Compilation option 2 (better)
# Using the ::quartus::flow package, and execute_flow command will
# export_assignments automatically and be equivalent to the steps
# outlined in compilation option 1
load_package flow
execute_flow -compile

# Close Project
project_close
```

Custom options are available to target other EDA tools. For custom EDA configurations, you can change the individual EDA interface options by making additional assignments.



For a complete list of each EDA setting line available, refer to the Quartus II Help.

Using LogicLock Regions

You can use Tcl commands to work with LogicLock™ regions. The following examples show how to export and import LogicLock regions for use in other designs. The examples use the files in the LogicLock tutorial design.



For additional information about the LogicLock design methodology, see the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

To compile a design and export LogicLock regions, follow these steps:

1. Create a project and add assignments.
2. Assign virtual pins.
3. Create a LogicLock region.
4. Assign a design entity to the region.
5. Compile the project.
6. Back-annotate the region.
7. Export the region.

Example 3–5 shows a script that creates a project called **lockmult**, and makes all the required assignments to compile the project. Next, the script compiles the project, back-annotates the design, and exports the LogicLock region. The script uses a procedure called **assign_virtual_pins**, which is described after the example. Use the **quartus_cdb** executable to run this script.

Example 3–5. LogicLock Export Script

```
load_package flow
load_package logiclock
load_package backannotate

project_new lockmult -overwrite
set_global_assignment -name BDF_FILE pipemult.bdf
set_global_assignment -name FAMILY Cyclone
set_global_assignment -name DEVICE EP1C6F256C6
set_global_assignment -name TOP_LEVEL_ENTITY pipemult

# These two assignments cause the Quartus II software
# to generate a VQM file for the logic in the LogicLock
# region. The VQM file is imported into the top-level
# design.
set_global_assignment -name \
   LOGICLOCK_INCREMENTAL_COMPILE_FILE pipemult.vqm
set_global_assignment -name \
   LOGICLOCK_INCREMENTAL_COMPILE_ASSIGNMENT ON

create_base_clock -fmax 200MHz -target clk clk_200
assign_virtual_pins { clk }
#Prepare LogicLock data structures before
#LogicLock-related commands.
initialize_logiclock

# Create a region named lockmult and assign pipemult
# to it.
# The region is auto-sized and floating.
set_logiclock -region lockmult -auto_size true \
-floating true
set_logiclock_contents -region lockmult -to pipemult
execute_flow -compile

# Back annotate the LogicLock Region and export a QSF
logiclock_back_annotate -region lockmult -lock
logiclock_export -file_name pipemult.qsf


uninitialize_logiclock
project_close
```

The **assign_virtual_pins** command is a procedure that makes virtual pin assignments to all bottom-level design pins, except for signals specified as arguments to the procedure. The procedure is defined in Example 3–6.

Example 3–6. assign_virtual_pins Procedure

```
proc assign_virtual_pins { skips } {

   # Analysis and elaboration must be run first to get the pin names
   execute_flow -analysis_and_elaboration

   # Get all pin names as a collection.

   set name_ids [get_names -filter * -node_type pin]
   foreach_in_collection name_id $name_ids {
      # Get the hierarchical path name of the pin.
      set hname [get_name_info -info full_path $name_id]
      #Skip the virtual pin assignment if the
      #pin is in the list of signals to be skipped.
      if {[lsearch -exact $skips $hname] == -1} {
         post_message "Setting VIRTUAL_PIN on $hname"
         set_instance_assignment -to $hname -name VIRTUAL_PIN ON
      } else {
         post_message "Skipping VIRTUAL_PIN for $hname"
      }
   }
}
```

When the script runs, it generates a netlist file named **pipemult.vqm**, and a Quartus II Settings File named **pipemult.qsf**, which contains the back-annotated assignments. You can then import the LogicLock region in another design. This example uses the top-level design in the **topmult** directory.

To import it four times in the top-level LogicLock tutorial design, follow these steps:

1. Create the top-level project.
2. Add assignments.
3. Elaborate the design.
4. Import the LogicLock constraints.
5. Compile the project.

Example 3–7 shows a script that demonstrates the previous steps.

Example 3–7. LogicLock Import Script

```
load_package flow
load_package logiclock

project_new topmult -overwrite
set_global_assignment -name BDF_FILE topmult.bdf
set_global_assignment -name VQM_FILE pipemult.vqm
set_global_assignment -name FAMILY Cyclone
set_global_assignment -name DEVICE EP1C6F256C6
create_base_clock -fmax 200MHz -target clk clk_200

# The LogicLock region will be used four times
# in the top-level design. These assignments
# specify that the back-annotated assignments in
# the QSF will be applied to the four entities
# in the top-level design.
set_instance_assignment -name LL_IMPORT_FILE pipemult.qsf \
    -to pipemult:inst
set_instance_assignment -name LL_IMPORT_FILE pipemult.qsf \
    -to pipemult:inst1
set_instance_assignment -name LL_IMPORT_FILE pipemult.qsf \
    -to pipemult:inst2
set_instance_assignment -name LL_IMPORT_FILE pipemult.qsf \
    -to pipemult:inst3

execute_flow -analysis_and_elaboration
initialize_logiclock
logiclock_import
uninitialize_logiclock
execute_flow -compile
project_close
```



For additional information about the LogicLock design methodology, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Compiling Designs

You can run the Quartus II command-line executables from Tcl scripts. Use the included **flow** package to run various Quartus II compilation flows, or run each executable directly.

The flow Package

The **flow** package includes two commands for running Quartus II command-line executables, either individually or together in standard compilation sequence. The **execute_module** command allows you to run

an individual Quartus II command-line executable. The **execute_flow** command allows you to run some or all of the modules in commonly-used combinations.

Altera recommends using the **flow** package instead of using system calls to run compiler executables.

Another way to run a Quartus II command-line executable from the Tcl environment is by using the **qexec** Tcl command, a Quartus II implementation of the Tcl **exec** command. For example, to run the Quartus II technology mapper on a given project, type:

```
qexec "quartus_map <project_name>" ↵
```

When you use the **qexec** command to compile a design, assignments made in the Tcl script (or from the Tcl shell) are not exported to the project database and settings file before compilation. Use the **export_assignments** command or compile the project with commands in the **flow** package to ensure assignments are exported to the project database and settings file.



You should use the **qexec** command to make system calls.

You can also run executables directly in a Tcl shell, using the same syntax as at the system command prompt. For example, to run the Quartus II technology mapper on a given project, type the following at the Tcl shell prompt:

```
quartus_map <project_name> ↵
```

Reporting

Once compilation finishes, it is sometimes necessary to extract information from the report to evaluate the results. For example, you may need to know how many device resources the design uses, or whether it meets your performance requirements. The Quartus II Tcl API provides easy access to report data so you don't have to write scripts to parse the text report files.

Use the commands that access report data one row at a time, or one cell at a time. If you know the exact cell or cells you want to access, use the **get_report_panel_data** command and specify the row and column names (or *x* and *y* coordinates) and the name of the appropriate report panel. You may often search for data in a report panel. To do this, use a loop that reads the report one row at a time with the **get_report_panel_row** command.

Column headings in report panels are in row 0. If you use a loop that reads the report one row at a time, you can start with row 1 to skip the row with column headings. The **get_number_of_rows** command returns the number of rows in the report panel, including the column heading row. Because the number of rows includes the column heading row, your loop should continue as long as the loop index is less than the number of rows, as illustrated in the following example.

Report panels are hierarchically arranged, and each level of hierarchy is denoted by the string "||" in the panel name. For example, the name of the Fitter Settings report panel is **Fitter||Fitter Settings** because it is in the Fitter folder. Panels at the highest hierarchy level do not use the "||" string. For example, the Flow Settings report panel is named **Flow Settings**.

Example 3–8 shows code that prints a list of all report panel names in your project.

Example 3–8. Print All Report Panel Names

```
set panel_names [get_report_panel_names]
foreach panel_name $panel_names {
   post_message "$panel_name"
}
```

Example 3–9 prints the number of failing paths in each clock domain in your design. It uses a loop to access each row of the **Timing Analyzer Summary** report panel. Clock domains are listed in the column named **Type** with the format `Clock Setup:'`*<clock name>*`'`. Other summary information is listed in the **Type** column as well. If the **Type** column matches the pattern "`Clock Setup*`", the script prints the number of failing paths listed in the column named **Failed Paths**.

Example 3–9. Print Number of Failing Paths per Clock

```
load_package report
project_open my-project
load_report
set report_panel_name "Timing Analyzer||Timing Analyzer Summary"
set num_rows [get_number_of_rows -name $report_panel_name]

# Get the column indices for the Type and Failed Paths columns
set type_column [get_report_panel_column_index -name \
    $report_panel_name "Type"]
set failed_paths_column [get_report_panel_column_index -name \
    $report_panel_name "Failed Paths"]

# Go through each line in the report panel
for {set i 1} {$i < $num_rows} {incr i} {

    # Get the row of data, then the type of summary
    # information in the row, and the number of failed paths
    set report_row [get_report_panel_row -name \
    $report_panel_name -row $i]
    set row_type [lindex $report_row $type_column]
    set failed_paths [lindex $report_row $failed_paths_column]
    if { [string match "Clock Setup*" $row_type] } {
        puts "$row_type has $failed_paths failing paths"
    }
}
unload_report
```

Creating CSV Files for Excel

The Microsoft Excel software is sometimes used to view or manipulate timing analysis results. You can create a CSV file to import into Excel with data from any Quartus II report. Example 3–10 shows a simple way to create a CSV file with data from a timing analysis panel in the report. You could modify the script to use command-line arguments to pass in the name of the project, report panel, and output file to use.

Example 3–10. Create CSV Files from Reports

```
load_package report
project_open my-project

load_report

# This is the name of the report panel to save as a CSV file
set panel_name "Timing Analyzer||Clock Setup: 'clk'"
set csv_file "output.csv"

set fh [open $csv_file w]
set num_rows [get_number_of_rows -name $panel_name]

# Go through all the rows in the report file, including the
# row with headings, and write out the comma-separated data
for { set i 0 } { $i < $num_rows } { incr i } {
    set row_data [get_report_panel_row -name $panel_name \
        -row $i]
    puts $fh [join $row_data ","]
}

close $fh
unload_report
```

Short Option Names

Beginning with version 6.0 of the Quartus II software, you can use short versions of command options, as long as they distinguish between the command's options. For example, the **project_open** command supports two options: `-current_revision` and `-revision`. You can use any of the following shortened versions of the `-revision` option: `-r`, `-re`, `-rev`, `-revi`, `-revis`, and `-revisio`. You can use an option as short as `-r` because no other option starts with the same letters as revision. However, the **report_timing** command includes the options `-recovery` and `-removal`. You cannot use `-r` or `-re` to shorten either of those options, because they do not uniquely distinguish between either option. You could use `-rec` or `-rem`.

Timing Analysis

The Quartus II software includes comprehensive Tcl APIs for both the Classic and TimeQuest analyzers. This section includes simple and advanced script examples for the Classic analyzer and introductory scripting information about the TimeQuest Tcl API.

Classic Timing Analysis

The following example script uses the **quartus_tan** executable to perform a timing analysis on the **fir_filter** tutorial design.

The **fir_filter** design is a two-clock design that requires a base clock and a relative clock relationship for timing analysis. This script first does an analysis of the two-clock relationship and checks for t_{SU} slack between `clk` and `clkx2`. The first pass of timing analysis discovers a negative slack for one of the clocks. The second part of the script adds a multicycle assignment from `clk` to `clkx2` and re-analyzes the design as a multi-clock, multicycle design.

The script does not recompile the design with the new timing assignments, and timing-driven compilation is not used in the synthesis and placement of this design. New timing assignments are added only for the timing analyzer to analyze the design with the **create_timing_netlist** and **report_timing** Tcl commands.



You must compile the project before running the script example shown in Example 3–11.

Example 3–11. Classic Timing Analysis

```
# This Tcl file is to be used with quartus_tan.exe
# This Tcl file will do the Quartus II tutorial fir_filter design
# timing analysis portion by making a global timing assignment and
# creating multi-clock assignments and run timing analysis
# for a multi-clock, multi-cycle design
# set the project_name to fir_filter
# set the revision_name to filtref
set project_name fir_filter
set revision_name filtref

# open the project
# project_name is the project name
project_open -revision $revision_name $project_name;

# Doing TAN tutorial steps this section re-runs the timing
# analysis module with multi-clock and multi-cycle settings
#------ Make timing assignments ------#

#Specifying a global FMAX requirement (tan tutorial)
set_global_assignment -name FMAX_REQUIREMENT 45.0MHz
set_global_assignment -name CUT_OFF_IO_PIN_FEEDBACK ON

# create a base reference clock "clocka" and specifies the
# following:
#   BASED_ON_CLOCK_SETTINGS = clocka;
#   INCLUDE_EXTERNAL_PIN_DELAYS_IN_FMAX_CALCULATIONS = OFF;
#   FMAX_REQUIREMENT = 50MHZ;
#   DUTY_CYCLE = 50;
# Assigns the reference clocka to the pin "clk"
create_base_clock -fmax 50MHZ -duty_cycle 50 clocka -target clk
```

```
# creates a relative clock "clockb" based on reference clock
# "clocka" with the following settings:
#   BASED_ON_CLOCK_SETTINGS = clocka;
#   MULTIPLY_BASE_CLOCK_PERIOD_BY = 1;
#   DIVIDE_BASE_CLOCK_PERIOD_BY = 2;clock period is half the base clk
#   DUTY_CYCLE = 50;
#   OFFSET_FROM_BASE_CLOCK = 500ps;The offset is .5 ns (or 500 ps)
#   INVERT_BASE_CLOCK = OFF;
# Assigns the reference clock to pin "clkx2"
create_relative_clock -base_clock clocka -duty_cycle 50\
-divide 2 -offset 500ps -target clkx2 clockb

# create new timing netlist based on new timing settings
create_timing_netlist

# does an analysis for clkx2
# Limits path listing to 1 path
# Does clock setup analysis for clkx2
report_timing -npaths 1 -clock_setup -file setup_multiclock.tao

# The output file will show a negative slack for clkx2 when only
# specifying a multi-clock design. The negative slack was created
# by the 500 ps offset from the base clock

# removes old timing netlist to allow for creation of a new timing
# netlist for analysis by report_timing
delete_timing_netlist

# adding a multi-cycle setting corrects the negative slack by adding a
# multicycle assignment to clkx2 to allow for more set-up time
set_multicycle_assignment 2 -from clk -to clkx2

# create a new timing netlist based on additional timing
# assignments create_timing_netlist

# outputs the result to a file for clkx2 only
report_timing -npaths 1 -clock_setup -clock_filter clkx2 \
 -file clkx2_setup_multicycle.tao
# The new output file will show a positive slack for the clkx2
project_close
```

Advanced Classic Timing Analysis

There may be times when the commands available for timing analysis reporting do not provide access to specific data you need. The **advanced_timing** package provides commands to access the data structures representing the timing netlist for your design. These commands provide low-level details about timing delays, node fan-in and fan-out, and timing data. Writing procedures to traverse the timing netlist and extract information gives you the most control to get exactly the data you need.

The timing netlist is represented with a graph, which is a collection of nodes and edges. Nodes represent elements in your design such as registers, combinational nodes, pins, and clocks. Edges connect the nodes and represent the connections between the logic in your design. Edges and nodes have unique positive integer IDs that identify them in the timing netlist. All the commands for getting information about the timing netlist use node and edge IDs instead of text-based names.

As an example of how to use the **advanced_timing** package, Example 3–12 shows one way to show the register-to-pin delays from all registers that drive the pins of an output bus. Specify the name of an output bus (for example, `address`), and the script prints out the names of all registers driving the pins of the bus and the delays from registers to pins.

Example 3–12. Report Register-to-Pin Delays

```
load_package advanced_timing
package require cmdline

# This procedure returns a list of IDs for pins with names
# that match the bus name passed in
proc find { bus_name } {

    set to_return [list]

    foreach_in_collection node_id [get_timing_nodes -type pin] {
        set node_name [get_timing_node_info -info name $node_id]
        if { [string match $bus_name* $node_name] } {
            lappend to_return $node_id
        }
    }
    return $to_return
}

# Required arguments for the script are the name of the project and
# revision, as well as the name of the bus to analyze
set options {\
    { "project.arg" "" "Project name" } \
    { "revision.arg" "" "Revision name" } \
    { "bus_name.arg" "" "Name of the bus to get timing data for" } \
}
array set opts [::cmdline::getoptions quartus(args) $options]

project_open $opts(project) -revision $opts(revision)

# The timing netlist must be created before accessing it.
create_timing_netlist

# This creates a data structure with additional timing data
create_p2p_delays

# Walk through each pin in the specified bus
foreach pin_id [find $opts(bus_name)] {
    set pin_name [get_timing_node_info -info name $pin_id]
```

```
    puts "$pin_name source registers and delays"
    # The get_delays_from_keepers command returns a list of all the
    # non-combinational nodes in the design that fan in to the
    # specified timing node, with the associated delays.
    foreach data [get_delays_from_keepers $pin_id] {
        set source_node [lindex $data 0]
        set max_delay [lindex $data 1]
        set source_node_name \
[get_timing_node_info -info name $source_node]
        puts " $source_node_name $max_delay"
    }
}
project_close
```

Type the command shown in Example 3–13 at a system command prompt to run this script.

Example 3–13.

```
quartus_tan -t script.tcl -project fir_filter
       -revision filtref -bus_name yn_out ↵
```

TimeQuest Timing Analysis

The Quartus II TimeQuest Timing Analyzer includes support for SDC commands in the **::quartus::sdc** package.



Refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* for detailed information about how to perform timing analysis with the Quartus II TimeQuest Timing Analyzer.

TimeQuest Scripting

In versions of the Quartus II software before 6.0, the **::quartus::project** Tcl package contained the following SDC-like commands for making timing assignments:

- **create_base_clock**
- **create_relative_clock**
- **get_clocks**
- **set_clock_latency**
- **set_clock_uncertainty**
- **set_input_delay**
- **set_multicycle_assignment**
- **set_output_delay**
- **set_timing_cut_assignment**

These commands are not SDC-compliant. Beginning with version 6.0, these commands are in a new package named **::quartus::timing_assignment**. To ensure backwards compatibility with existing Tcl scripts, the **::quartus::timing_assignment** package is loaded by default in the following executables:

- **quartus**
- **quartus_sh**
- **quartus_cdb**
- **quartus_sim**
- **quartus_stp**
- **quartus_tan**

The **::quartus::timing_assignment** package is not loaded by default in the **quartus_sta** executable. The **::quartus::sdc** Tcl package includes SDC-compliant versions of the commands listed above. The package is available only in the **quartus_sta** executable, and it is loaded by default.

Automating Script Execution

Beginning with version 4.0 of the Quartus II software, you can configure scripts to run automatically at various points during compilation. Use this capability to automatically run scripts that perform custom reporting, make specific assignments, and perform many other tasks.

The following three global assignments control when a script is run automatically:

- `PRE_FLOW_SCRIPT_FILE` —before a flow starts
- `POST_MODULE_SCRIPT_FILE` —after a module finishes
- `POST_FLOW_SCRIPT_FILE` —after a flow finishes

The `POST_FLOW_SCRIPT_FILE` and `POST_MODULE_SCRIPT_FILE` assignments are supported beginning in version 4.0, and the `PRE_FLOW_SCRIPT_FILE` assignment is supported beginning in version 4.1.

A module is a Quartus II executable that performs one step in a flow. For example, two modules are Analysis and Synthesis (**quartus_map**) and timing analysis (**quartus_tan**).

A flow is a series of modules that the Quartus II software runs with predefined options. For example, compiling a design is a flow that typically consists of the following steps (performed by the indicated module):

1. Analysis and synthesis (**quartus_map**)
2. Fitter (**quartus_fit**)

3. Assembler (**quartus_asm**)

4. Timing Analyzer (**quartus_tan**)

Other flows are described in the help for the **execute_flow** Tcl command. In addition, many commands in the Processing menu of the Quartus II GUI correspond to this design flow.

Making the Assignment

To make an assignment to automatically run a script, add an assignment with the following form to your project's Quartus II Settings File:

```
set_global_assignment -name <assignment name> \
    <executable>:<script name>
```

The assignment name is one of the following:

- `PRE_FLOW_SCRIPT_FILE`
- `POST_MODULE_SCRIPT_FILE`
- `POST_FLOW_SCRIPT_FILE`

The executable is the name of a Quartus II command-line executable that includes a Tcl interpreter.

- **quartus_cdb**
- **quartus_sh**
- **quartus_sim**
- **quartus_sta**
- **quartus_stp**
- **quartus_tan**

The script name is the name of your Tcl script.

Script Execution

The Quartus II software runs the scripts as shown in Example 3–14.

Example 3–14.

```
<executable> -t <script name> <flow or module name> <project name> <revision name>
```

The first argument passed in the `argv` variable (or `quartus(args)` variable) is the name of the flow or module being executed, depending on the assignment you use. The second argument is the name of the project, and the third argument is the name of the revision.

When you use the POST_MODULE_SCRIPT_FILE assignment, the specified script is automatically run after every executable in a flow. You can use a string comparison with the module name (the first argument passed in to the script) to isolate script processing to certain modules.

Execution Example

Example 3–15 illustrates how automatic script execution works in a complete flow, assuming you have a project called **top** with a current revision called **rev_1**, and you have the following assignments in the Quartus II Settings File for your project.

Example 3–15.

```
set_global_assignment -name PRE_FLOW_SCRIPT_FILE quartus_sh:first.tcl
set_global_assignment -name POST_MODULE_SCRIPT_FILE quartus_sh:next.tcl
set_global_assignment -name POST_FLOW_SCRIPT_FILE quartus_sh:last.tcl
```

When you compile your project, the PRE_FLOW_SCRIPT_FILE assignment causes the following command to be run before compilation begins:

```
quartus_sh -t first.tcl compile top rev_1
```

Next, the Quartus II software starts compilation with analysis and synthesis, performed by the **quartus_map** executable. After the analysis and synthesis finishes, the POST_MODULE_SCRIPT_FILE assignment causes the following command to be run:

```
quartus_sh -t next.tcl quartus_map top rev_1
```

Then, the Quartus II software continues compilation with the Fitter, performed by the **quartus_fit** executable. After the Fitter finishes, the POST_MODULE_SCRIPT_FILE assignment runs the following command:

```
quartus_sh -t next.tcl quartus_fit top rev_1
```

Corresponding commands are run after the other stages of the compilation. Finally, after the compilation is over, the POST_FLOW_SCRIPT_FILE assignment runs the following command:

```
quartus_sh -t last.tcl compile top rev_1
```

Controlling Processing

The POST_MODULE_SCRIPT_FILE assignment causes a script to run after every module. Because the same script is run after every module, you may need to include some conditional statements that restrict processing in your script to certain modules.

For example, if you want a script to run only after timing analysis, you should include a conditional test like the one shown in Example 3–16. It checks the flow or module name passed as the first argument to the script and executes code when the module is quartus_tan.

Example 3–16. Restrict Processing to a Single Module

```
set module [lindex $quartus(args) 0]

if [string match "quartus_tan" $module] {

    # Include commands here that are run
    # after timing analysis
    # Use the post-message command to display
    # messages
    post_message "Running after timing analysis"
}
```

Displaying Messages

Because of the way the Quartus II software runs the scripts automatically, you must use the **post_message** command to display messages, instead of the **puts** command. This requirement applies only to scripts that are run by the three assignments listed in "Automating Script Execution" on page 3–30.



Refer to "Using the post_message Command" on page 3–35 for more information about this command.

Other Scripting Features

The Quartus II Tcl API includes other general-purpose commands and features described in this section.

Natural Bus Naming

Beginning with version 4.2, the Quartus II software supports natural bus naming. Natural bus naming means that square brackets used to specify bus indexes in hardware description languages do not have to be escaped to prevent Tcl from interpreting them as commands. For example, one

signal in a bus named address can be identified as `address[0]` instead of `address\[0\]`. You can take advantage of natural bus naming when making assignments, as in Example 3–17.

Example 3–17. Natural Bus Naming

```
set_location_assignment -to address[10] Pin_M20
```

The Quartus II software defaults to natural bus naming. You can turn off natural bus naming with the **disable_natural_bus_naming** command. For more information about natural bus naming, type `enable_natural_bus_naming -h` ↵ at a Quartus II Tcl prompt.

Using Collection Commands

Some Quartus II Tcl functions return very large sets of data that would be inefficient as Tcl lists. These data structures are referred to as collections and the Quartus II Tcl API uses a collection ID to access the collection. There are two Quartus II Tcl commands for working with collections, **foreach_in_collection** and **get_collection_size**. Use the **set** command to assign a collection ID to a variable.



For information about which Quartus II Tcl commands return collection IDs, see the Quartus II Help and search for the **foreach_in_collection** command.

The foreach_in_collection Command

The **foreach_in_collection** command is similar to the **foreach** Tcl command. Use it to iterate through all elements in a collection. Example 3–18 prints all instance assignments in an open project.

Example 3–18. Using Collection Commands

```
set all_instance_assignments [get_all_instance_assignments -name *]
foreach_in_collection asgn $all_instance_assignments {
    # Information about each assignment is
    # returned in a list. For information
    # about the list elements, refer to Help
    # for the get-all-instance-assignments command.
    set to [lindex $asgn 2]
    set name [lindex $asgn 3]
    set value [lindex $asgn 4]
    puts "Assignment to $to: $name = $value"
}
```

The get_collection_size Command

Use the **get_collection_size** command to get the number of elements in a collection. Example 3–19 prints the number of global assignments in an open project.

Example 3–19. get_collection_size Command

```
set all_global_assignments [get_all_global_assignments -name *]
set num_global_assignments [get_collection_size $all_global_assignments]
puts "There are $num_global_assignments global assignments in your project"
```

Using the post_message Command

To print messages that are formatted like Quartus II software messages, use the **post_message** command. Messages printed by the **post_message** command appear in the **System** tab of the Messages window in the Quartus II GUI, and are written to standard at when scripts are run. Arguments for the **post_message** command include an optional message type and a required message string.

The message type can be one of the following:

- info (default)
- extra_info
- warning
- critical_warning
- error

If you do not specify a type, Quartus II software defaults to `info`.

When you are using the Quartus II software in Windows, you can color code messages displayed at the system command prompt with the **post_message** command. Add the following line to your **quartus2.ini** file:

```
DISPLAY_COMMAND_LINE_MESSAGES_IN_COLOR = on
```

Example 3–20 shows how to use the **post_message** command.

Example 3–20. post_message command

```
post_message -type warning "Design has gated clocks"
```

Accessing Command-Line Arguments

Many Tcl scripts are designed to accept command-line arguments, such as the name of a project or revision. The global variable `quartus(args)` is a list of the arguments typed on the command-line following the name of the Tcl script. Example 3–21 shows code that prints all of the arguments in the `quartus(args)` variable.

Example 3–21. Simple Command-Line Argument Access

```
set i 0
foreach arg $quartus(args) {
   puts "The value at index $i is $arg"
   incr i
}
```

If you copy the script in the previous example to a file named **print_args.tcl**, it displays the following output when you type the command shown in Example 3–22 at a command prompt.

Example 3–22. Passing Command-Line Arguments to Scripts

```
quartus_sh -t print_args.tcl my_project 100MHz ↵
The value at index 0 is <my_project>
The value at index 1 is 100MHz
```

Beginning with version 4.1, the Quartus II software supports the Tcl variables `argv`, `argc`, and `argv0` for command-line argument access. Table 3–7 shows equivalent information for earlier versions of the software.

Table 3–7. Quartus II Support for Tcl Variables

Beginning with Version 4.1	Equivalent Support in Previous Software Versions
argc	`llength $quartus(args)`
argv	`quartus(args)`
argv0	`info nameofexecutable`

Using the cmdline Package

You can use the **cmdline** package included with the Quartus II software for more robust and self-documenting command-line argument passing. The **cmdline** package supports command-line arguments with the form *-<option> <value>*.

Example 3–23 uses the **cmdline** package.

Example 3–23. cmdline Package

```
package require cmdline
variable ::argv0 $::quartus(args)
set options {\
   { "project.arg" "" "Project name" } \
   { "frequency.arg" ""  "Frequency" } \
}
set usage "You need to specify options and values"

array set optshash [::cmdline::getoptions ::argv $options $usage]
puts "The project name is $optshash(project)"
puts "The frequency is $optshash(frequency)"
```

If you save those commands in a Tcl script called **print_cmd_args.tcl** you see the following output when you type the command shown in Example 3–24 at a command prompt.

Example 3–24. Passing Command-Line Arguments for Scripts

```
quartus_sh -t print_cmd_args.tcl -project my_project -frequency 100MHz ↵
The project name is <my_project>
The frequency is 100MHz
```

Virtually all Quartus II Tcl scripts must open a project. Example 3–25 opens a project, and you can optionally specify a revision name. The example checks whether the specified project exists. If it does, the example opens the current revision, or the revision you specify.

Example 3–25. Full-Featured Method to Open Projects

```
package require cmdline
variable ::argv0 $::quartus(args)
set options { \
{ "project.arg" "" "Project Name" } \
{ "revision.arg" "" "Revision Name" } \
}
array set optshash [::cmdline::getoptions ::argv0 $options]

# Ensure the project exists before trying to open it
if {[project_exists $optshash(project)]} {

    if {[string equal "" $optshash(revision)]} {

        # There is no revision name specified, so default
        # to the current revision
        project_open $optshash(project) -current_revision
    } else {

        # There is a revision name specified, so open the
        # project with that revision
        project_open $optshash(project) -revision \
            $optshash(revision)
    }
} else {
    puts "Project $optshash(project) does not exist"
    exit 1
}
# The rest of your script goes here
```

If you do not require this flexibility or error checking, you can use just the **project_open** command, as shown in Example 3–26.

Example 3–26. Simple Method to Open Projects

```
set proj_name [lindex $argv 0]
project_open $proj_name
```



For more information about the **cmdline** package, refer to the documentation for the package at <*Quartus II installation directory*> **/common/tcl/packages**.

Using the Quartus II Tcl Shell in Interactive Mode

This section presents an example of using the **quartus_sh** interactive shell to make some project assignments and compile the finite impulse response (FIR) filter tutorial project. This example assumes that you already have the FIR filter tutorial design files in a project directory.

To begin, run the interactive Tcl shell. The command and initial output are shown in Example 3–27.

Example 3–27. Interactive Tcl Shell

```
tcl> quartus_sh -s
tcl> Info: ************************************************************************
Info: Running Quartus II Shell
    Info: Version 7.1 Full Version
    Info: Copyright (C) 1991-2007 Altera Corporation. All rights reserved.
    Info: Your use of Altera Corporation's design tools, logic functions
    Info: and other software and tools, and its AMPP partner logic
    Info: functions, and any output files any of the foregoing
    Info: (including device programming or simulation files), and any
    Info: associated documentation or information are expressly subject
    Info: to the terms and conditions of the Altera Program License
    Info: Subscription Agreement, Altera MegaCore Function License
    Info: Agreement, or other applicable license agreement, including,
    Info: without limitation, that your use is for the sole purpose of
    Info: programming logic devices manufactured by Altera and sold by
    Info: Altera or its authorized distributors.  Please refer to the
    Info: applicable agreement for further details.
    Info: Processing started: Wed Apr 04 12:24:13 2007
Info: ************************************************************************
Info: The Quartus II Shell supports all TCL commands in addition
Info: to Quartus II Tcl commands. All unrecognized commands are
Info: assumed to be external and are run using Tcl's "exec"
Info: command.
Info: - Type "exit" to exit.
Info: - Type "help" to view a list of Quartus II Tcl packages.
Info: - Type "help <package name>" to view a list of Tcl commands
Info:   available for the specified Quartus II Tcl package.
Info: - Type "help -tcl" to get an overview on Quartus II Tcl usages.
Info: ************************************************************************

tcl>
```

Create a new project called **fir_filter**, with a revision called **filtref** by typing the following command at a Tcl prompt:

```
project_new -revision filtref fir_filter ↵
```



If the project file and project name are the same, the Quartus II software gives the revision the same name as the project.

Because the revision named **filtref** matches the top-level file, all design files are automatically picked up from the hierarchy tree.

Next, set a global assignment for the device with the following command:

```
set_global_assignment -name family Cyclone ↵
```

To learn more about assignment names that you can use with the `-name` option, refer to the Quartus II Help.

For assignment values that contain spaces, the value should be enclosed in quotation marks.

To quickly compile a design, use the `::quartus::flow` package, which properly exports the new project assignments and compiles the design using the proper sequence of the command-line executables. First, load the package:

```
load_package flow ↵
```

It returns the following:

```
1.0
```

For additional help on the `::quartus::flow` package, view the command-line help at the Tcl prompt by typing:

```
help -pkg  ::quartus::flow ↵
```

Example 3–28 shows an alternative command and the resulting output.

Example 3–28. Help Output

```
tcl> help -pkg flow
------------------------------------------------------------------
-------------------------
Tcl Package and Version:
-------------------------
::quartus::flow 1.0
------------
Description:
------------
        This package contains the set of Tcl functions
        for running flows or command-line executables.
-------------
Tcl Commands:
-------------
        execute_flow
        execute_module
------------------------------------------------------------------
```

This help display gives information about the flow package and the commands that are available with the package.To learn about the options available for the **execute_flow** Tcl command, type the following command at a Tcl prompt:

```
execute_flow -h ↵
```

To view additional information and example usage type the following command at a Tcl prompt:

```
execute_flow -long_help ↵
```

or

```
help -cmd execute_flow ↵
```

To perform a full compilation of the FIR filter design, use the `execute_flow` command with the `-compile` option, as shown in Example 3–29.

Example 3–29.

```
tcl> execute_flow -compile ↵
Info:*******************************************************
Info: Running Quartus II Analysis & Synthesis
Info: Version 6.0 SJ Full Version
Info: Processing started: Tues Apr 04 09:30:47 2006
Info: Command: quartus_map --import_settings_files=on --
export_settings_files=of fir_filter -c filtref
.
.
.
Info: Writing report file filtref.tan.rpt
tcl>
```

This script compiles the FIR filter tutorial project, exporting the project assignments and running **quartus_map**, **quartus_fit**, **quartus_asm,** and **quartus_tan**. This sequence of events is the same as selecting **Start Compilation** from the Processing menu in the Quartus II GUI.

When you are finished with a project, close it using the **project_close** command as shown in Example 3–30.

Example 3–30.

```
project_close ↵
```

Then, to exit the interactive Tcl shell, type `exit ↵` at a Tcl prompt.

Quartus II Legacy Tcl Support

Beginning with the Quartus II software version 3.0, command-line executables do not support the Tcl commands used in previous versions of the Quartus II software. These commands are supported in the GUI with the Quartus II Tcl console or by using the **quartus_cmd** executable at the system command prompt. If you source Tcl scripts developed for an earlier version of the Quartus II software using either of these methods, the project assignments are ported to the project database and settings file. You can then use the command-line executables to process the resulting project. This may be necessary if you create a Tcl file using EDA tools that do not generate Tcl scripts for the most recent version of the Quartus II software.



You should create all new projects and Tcl scripts with the latest version of the Quartus II Tcl API.

Tcl Scripting Basics

The core Tcl commands support variables, control structures, and procedures. Additionally, there are commands for accessing the file system and network sockets, and running other programs. You can create platform-independent graphical interfaces with the Tk widget set.

Tcl commands are executed immediately as they are typed in an interactive Tcl shell. You can also create scripts (including this chapter's examples) as files and run them with a Tcl interpreter. A Tcl script does not need any special headers.

To start an interactive Tcl interpreter, type `quartus_sh -s` ↵ at a command prompt. The commands you type are executed immediately at the interpreter prompt. If you save a series of Tcl commands in a file, you can run it with a Tcl interpreter. To run a script named **myscript.tcl**, type `quartus_sh -t myscript.tcl` ↵ at a command prompt.

Hello World Example

The following shows the basic "Hello world" example in Tcl:

```
puts "Hello world"
```

Use double quotation marks to group the words `hello` and `world` as one argument. Double quotation marks allow substitutions to occur in the group. Substitutions can be simple variable substitutions, or the result of running a nested command, described in "Substitutions" on page 3–43. Use curly braces { } for grouping when you want to prevent substitutions.

Variables

Use the **set** command to assign a value to a variable. You do not have to declare a variable before using it. Tcl variable names are case-sensitive. Example 3–31 assigns the value `1` to the variable named `a`.

Example 3–31. Assigning Variables

```
set a 1
```

To access the contents of a variable, use a dollar sign before the variable name. Example 3–32 prints `"Hello world"` in a different way.

Example 3–32. Accessing Variables

```
set a Hello
set b world
puts "$a $b"
```

Substitutions

Tcl performs three types of substitution:

- Variable value substitution
- Nested command substitution
- Backslash substitution

Variable Value Substitution

Variable value substitution, as shown in Example 3–32, refers to accessing the value stored in a variable by using a dollar sign ("$") before the variable name.

Nested Command Substitution

Nested command substitution refers to how the Tcl interpreter evaluates Tcl code in square brackets. The Tcl interpreter evaluates nested commands, starting with the innermost nested command, and commands nested at the same level from left to right. Each nested command result is substituted in the outer command. Example 3–33 sets `a` to the length of the string `foo`.

Example 3–33. Command Substitution

```
set a [string length foo]
```

Backlash Substitution

Backslash substitution allows you to quote reserved characters in Tcl, such as dollar signs ("$") and braces ("[ ]"). You can also specify other special ASCII characters like tabs and new lines with backslash substitutions. The backslash character is the Tcl line continuation character, used when a Tcl command wraps to more than one line. Example 3–34 shows how to use the backslash character for line continuation.

Example 3–34. Backslash Substitution

```
set this_is_a_long_variable_name [string length "Hello \
    world."]
```

Arithmetic

Use the **expr** command to perform arithmetic calculations. Using curly braces ("{ }") to group the arguments of this command makes arithmetic calculations more efficient and preserves numeric precision. Example 3–35 sets `b` to the sum of the value in the variable `a` and the square root of `2`.

Example 3–35. Arithmetic with the expr Command

```
set a 5
set b [expr { $a + sqrt(2) }]
```

Tcl also supports boolean operators such as `&&` (AND), `||` (OR), `!` (NOT), and comparison operators such as `<` (less than), `>` (greater than), and `==` (equal to).

Lists

A Tcl list is a series of values. Supported list operations include creating lists, appending lists, extracting list elements, computing the length of a list, sorting a list, and more. Example 3–36 sets `a` to a list with three numbers in it.

Example 3–36. Creating Simple Lists

```
set a { 1 2 3 }
```

You can use the **lindex** command to extract information at a specific index in a list. Indexes are zero-based. You can use the index `end` to specify the last element in the list, or the index `end-<n>` to count from the end of the list. Example 3–37 prints the second element (at index 1) in the list stored in `a`.

Example 3–37. Accessing List Elements

```
puts [lindex $a 1]
```

The **llength** command returns the length of a list. Example 3–38 prints the length of the list stored in `a`.

Example 3–38. List Length

```
puts [llength $a]
```

The **lappend** command appends elements to a list. If a list does not already exist, the list you specify is created. The list variable name is not specified with a dollar sign. Example 3–39 appends some elements to the list stored in `a`.

Example 3–39. Appending to a List

```
lappend a 4 5 6
```

Arrays

Arrays are similar to lists except that they use a string-based index. Tcl arrays are implemented as hash tables. You can create arrays by setting each element individually or by using the **array set** command. To set an element with an index of `Mon` to a value of `Monday` in an array called `days`, use the following command:

```
set days(Mon) Monday
```

The `array set` command requires a list of index/value pairs. This example sets the array called days:

```
array set days { Sun Sunday Mon Monday Tue Tuesday \
   Wed Wednesday Thu Thursday Fri Friday Sat Saturday }
```

Example 3–40 shows how to access the value for a particular index.

Example 3–40. Accessing Array Elements

```
set day_abbreviation Mon
puts $days($day_abbreviation)
```

Use the **array names** command to get a list of all the indexes in a particular array. The index values are not returned in any specified order. Example 3–41 shows one way to iterate over all the values in an array.

Example 3–41. Iterating Over Arrays

```
foreach day [array names days] {
   puts "The abbreviation $day corresponds to the day \
name $days($day)"
}
```

Arrays are a very flexible way of storing information in a Tcl script and are a good way to build complex data structures.

Control Structures

Tcl supports common control structures, including **if-then-else** conditions and **for**, **foreach**, and **while** loops. The position of the curly braces as shown in the following examples ensures the control structure commands are executed efficiently and correctly. Example 3–42 prints whether the value of variable a positive, negative, or zero.

Example 3–42. If-Then-Else Structure

```
if { $a > 0 } {
   puts "The value is positive"
} elseif { $a < 0 } {
   puts "The value is negative"
} else {
   puts "The value is zero"
}
```

Example 3–43 uses a **for** loop to print each element in a list.

Example 3–43. For Loop

```
set a { 1 2 3 }
for { set i 0 } { $i < [llength $a] } { incr i } {
   puts "The list element at index $i is [lindex $a $i]"
}
```

Example 3–44 uses a **foreach** loop to print each element in a list.

Example 3–44. foreach Loop

```
set a { 1 2 3 }
foreach element $a {
    puts "The list element is $element"
}
```

Example 3–45 uses a **while** loop to print each element in a list.

Example 3–45. while Loop

```
set a { 1 2 3 }
set i 0
while { $i < [llength $a] } {
    puts "The list element at index $i is [lindex $a $i]"
    incr i
}
```

You do not need to use the **expr** command in boolean expressions in control structure commands because they invoke the **expr** command automatically.

Procedures

Use the **proc** command to define a Tcl procedure (known as a subroutine or function in other scripting and programming languages). The scope of variables in a procedure is local to the procedure. If the procedure returns a value, use the **return** command to return the value from the procedure. Example 3–46 defines a procedure that multiplies two numbers and returns the result.

Example 3–46. Simple Procedure

```
proc multiply { x y } {
    set product [expr { $x * $y }]
    return $product
}
```

Example 3–47 shows how to use the **multiply** procedure in your code. You must define a procedure before your script calls it, as shown below.

Example 3–47. Using a Procedure

```
proc multiply { x y } {
    set product [expr { $x * $y }]
    return $product
}
set a 1
set b 2
puts [multiply $a $b]
```

You should define procedures near the beginning of a script. If you want to access global variables in a procedure, use the **global** command in each procedure that uses a global variable. Example 3–48 defines a procedure that prints an element in a global list of numbers, then calls the procedure.

Example 3–48. Accessing Global Variables

```
proc print_global_list_element { i } {
    global my_data
    puts "The list element at index $i is [lindex $my_data $i]"
}
set my_data { 1 2 3}
print_global_list_element 0
```

File I/O

Tcl includes commands to read from and write to files. You must open a file before you can read from or write to it, and close it when the read and write operations are done. To open a file, use the **open** command; to close a file, use the **close** command. When you open a file, specify its name and the mode in which to open it. If you do not specify a mode, Tcl defaults to read mode. To write to a file, specify `w` for write mode as shown in Example 3–49.

Example 3–49. Open a File for Writing

```
set output [open myfile.txt w]
```

Tcl supports other modes, including appending to existing files and reading from and writing to the same file.

The **open** command returns a file handle to use for read or write access. You can use the **puts** command to write to a file by specifying a filehandle, as shown in Example 3–50.

Example 3–50. Write to a File

```
set output [open myfile.txt w]
puts $output "This text is written to the file."
close $output
```

You can read a file one line at a time with the **gets** command. Example 3–51 uses the **gets** command to read each line of the file and then prints it out with its line number.

Example 3–51. Read from a File

```
set input [open myfile.txt]
set line_num 1
while { [gets $input line] >= 0 } {
    # Process the line of text here
    puts "$line_num: $line"
    incr line_num
}
close $input
```

Syntax and Comments

Arguments to Tcl commands are separated by white space, and Tcl commands are terminated by a newline character or a semicolon. As shown in "Substitutions" on page 3–43, you must use backslashes when a Tcl command extends more than one line.

Tcl uses the hash or pound character (#) to begin comments. The # character must begin a comment. If you prefer to include comments on the same line as a command, be sure to terminate the command with a semicolon before the # character. Example 3–52 is a valid line of code that includes a **set** command and a comment.

Example 3–52. Comments

```
set a 1;# Initializes a
```

Without the semicolon, it would be an invalid command because the **set** command would not terminate until the new line after the comment.

The Tcl interpreter counts curly braces inside comments, which can lead to errors that are difficult to track down. Example 3–53 causes an error because of unbalanced curly braces.

Example 3–53. Unbalanced Braces in Comments

```
# if { $x > 0 } {
if { $y > 0 } {
    # code here
}
```

External References



For more information about using Tcl, refer to the following sources:

- *Practical Programming in Tcl and Tk*, Brent B. Welch
- *Tcl and the TK Toolkit*, John Ousterhout
- *Effective Tcl/TK Programming*, Michael McLennan and Mark Harrison
- Quartus II Tcl example scripts at http://www.altera.com/support/examples/tcl/tcl.html
- Tcl Developer Xchange at http://tcl.activestate.com/

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Script-Based Design for HardCopy Devices* chapter of the *HardCopy Handbook*
- *Volume 3: Verification* of the *Quartus II Handbook*

Document Revision History

Table 3–8 shows the revision history for this document.

Table 3–8. Document Revision History

Date / Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 3–50.	Updated for the Quartus II software version 7.2.
May 2007 v7.1.0	Updated for the Quartus II software version 7.1 release, including: • Added `sdc_ext` information in Table 3-1 • Added `quartus_staw` information in Table 3-2 • Added Referenced Documents • Added a mini-TOC in the Introduction.	Minor updates for the Quartus II software version 7.1 release.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Added revision history to the document.	—
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0. • Reorganized content. • Added the Quartus II TimeQuest Timing Analyzer feature.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.0.	—
August 2005 v5.0.1	Minor text changes.	—
May 2005 v5.0.0	Updated for the Quartus II software version 5.0.0.	—
Dec. 2004 v2.1	• Updates to tables and figures. • New functionality in the Quartus II software version 4.2.	—
Aug. 2004 v2.1	• Minor typographical corrections • Enhancements to example scripts.	—
June 2004 v2.0	• Updates to tables, figures. • New functionality in the Quartus II software version 4.1.	—
Feb. 2004 v1.0	Initial release.	—

4. Managing Quartus II Projects

QII52012-7.2.0

Introduction

FPGA designs once required just one or two engineers, but today's larger and more sophisticated FPGA designs are often developed by several engineers and are constantly changing throughout the project. To ensure efficient design coordination, designers must track their project changes.

To help designers manage their FPGA designs, the Quartus® II software provides the following capabilities:

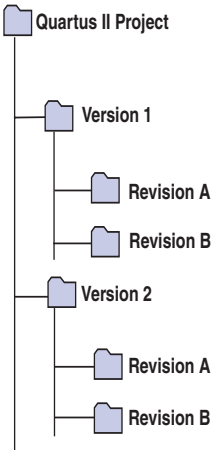
- Creating revisions
- Copying and archiving projects
- Making a version-compatible database
- Controlling message suppression

In the Quartus II software, a revision is one set of assignments and settings. A project typically has multiple revisions, with each revision having its own set of assignments and settings. Creating multiple revisions allows you to change assignments and settings while preserving the previous results.

A version is a Quartus II project that includes one set of design files and one or more revisions (assignments and settings for your project). Creating multiple versions with the Copy Project feature allows you to edit a copy of your design files while preserving the original functionality of your design.

The Quartus II Version-Compatible Database feature allows Quartus II databases to be compatible across different versions of the Quartus II software, which saves valuable design time by avoiding unnecessary compilations (Figure 4–1). This chapter also discusses how to migrate your projects from one computing platform to another as well as message suppression.

Figure 4–1. Quartus II Version-Compatible Database Structure



Creating a New Project

A Quartus II project contains all of the design files, settings files, and other files necessary for the successful compilation of your design. These files include two Quartus II settings files:

- Quartus II Project File (**.qpf**) containing the name of your project and all revisions of your project, described in "Using Revisions With Your Design" on page 4–3.
- Quartus II Settings File (**.qsf**) containing all assignments applied to your design including assignments to help fit your design and meet performance requirements. For more information on the Quartus II Settings File, refer to "Quartus II Settings File" on page 4–25.

To start a new Quartus II project, use the **New Project Wizard**. From the File menu, click the **New Project Wizard,** and add the following project information:

- Project name and directory
- Name of the top-level design entity
- Project files and user libraries
- Target device family and device
- EDA tool settings



For more information on user libraries, refer to "Specifying Libraries" on page 4–13 and "Specifying Libraries Using Scripts" on page 4–29.

Using Revisions With Your Design

The Revisions feature allows you to create a new set of assignments and settings for your design without losing your previous assignments and settings. This feature allows you to explore different assignments and settings for your design and then compare the results. You can use the Revisions feature in the following ways:

- Create a unique revision which is not based on a previous revision. Creating a unique revision lets you optimize a design for different fundamental reasons such as to optimize by area in one revision, then optimize for f_{MAX} in another revision. When you create a unique revision (a revision that is not based on an existing revision), all default settings are turned on.
- Create a revision based on an existing revision, but try new settings and assignments in the new revision. A new revision already includes all the assignments and settings applied in the previous revision. If you are not satisfied with the results in the new revision, you can revert to the original revision. You can compare revisions manually or with the Compare feature.

Creating and Deleting Revisions

All Quartus II assignments and settings are stored in the Quartus II Settings File. Each time you create a new revision of a project, the Quartus II software creates a new Quartus II Settings File and adds the name of the new revision to the list of revisions in the Quartus II Settings File.



The name of a new Quartus II Settings File matches the revision name.

You can manage revisions with the **Revisions** dialog box, which allows you to set the current revision, as well as create, delete, and compare revisions in a project.

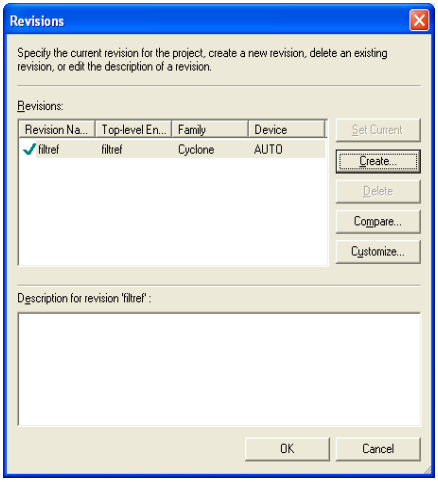
Creating a Revision

To create a revision, follow these steps:

1. If you have not already done so, create a new project or open an existing project. On the File menu, click **New Project Wizard** or **Open Project.**
2. On the Project menu, click **Revisions**.
3. To base the new revision on an existing revision for the current design, select the existing revision in the **Revisions** list.

4. Click **Create** (Figure 4–2).

Figure 4–2. Revisions Dialog Box



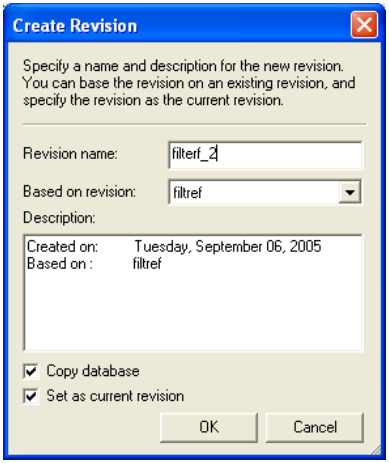
5. In the **Create Revision** dialog box (Figure 4–3), type the name of the new revision in the **Revisions name** box.

6. To base the new revision on an existing revision for the current design, if you did not select the correct revision in Step 3, select the revision in the **Based on revision** list (Figure 4–3).

   or

   To create a unique revision that is not based on an existing revision of the current design, select the "blank entry" in the **Based on revision** list.

Figure 4–3. Create Revision Dialog Box



7. Optionally, edit the description of the revision in the **Description** box (Figure 4–3).

8. Turn off the **Copy database** option if you do not want the new revision to contain the database information from the existing revision. The **Copy database** option is on by default.

   Copying the database allows you to view the results from the previous compilation while you are making changes to the settings of the new revision.

9. If you do not want to use the new revision immediately, turn off **Set as current revision.** The **Set as current revision** option is on by default.

10. Click **OK**.

Delete a Revision

To delete a revision, follow these steps:

1. If you have not already done so, open an existing project by clicking **Open Project on the File menu and selecting a Quartus II Project File.**

2. On the Project menu, click **Revisions**.

3. In the **Revisions** list, select the revision you want to delete and click **Delete**.



You cannot delete the current revision when it is active; you must first open another revision. For example, if revision A is currently active, you need to create (if the revision does not exist) and open revision B before you delete revision A.

Comparing Revisions

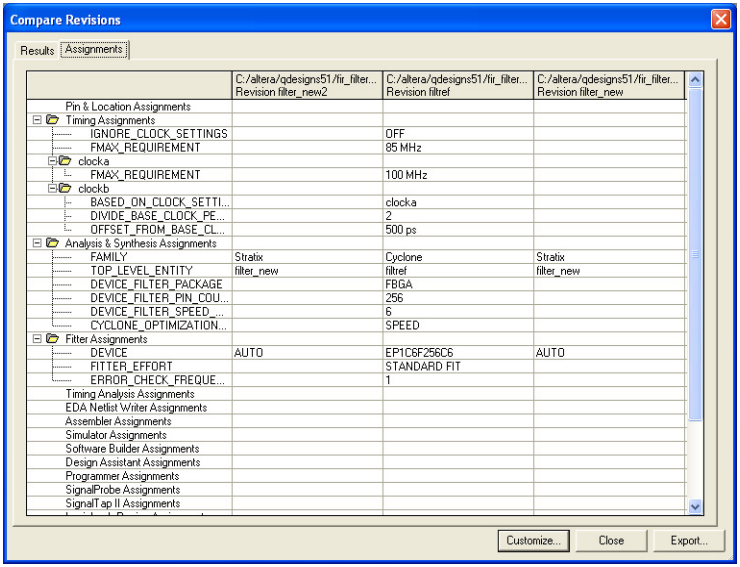
You can compare the results of multiple revisions side by side with the **Compare Revisions** dialog box.

1. On the Project menu, click **Revisions**.

2. In the **Revisions** dialog box, click **Compare** to compare all revisions in a single window.

The **Compare Revisions** dialog box (Figure 4–4) compares the results of each revision in three assignment categories: Analysis & Synthesis, Fitter, and Timing Analyzer.

Figure 4–4. Compare Revisions Dialog Box



In addition to viewing the results of each revision, you also can show the assignments for each revision. Click the **Assignments** tab in the **Compare Revisions** dialog box to view all assignments applied to each revision (Figure 4–4). To export both **Results** and **Assignments** for your revisions, click on **Export.** When the dialog box appears, enter the name of the Comma-Separated Value (.**csv**) file into which the software will export the data when you click **OK**. Gain better understanding of how different optimization options affect your design by viewing the results of each revision and their assignments.

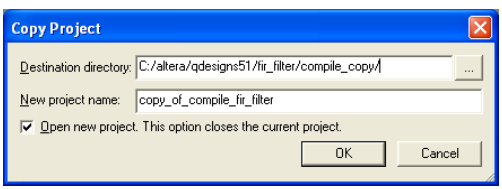
Creating Different Versions of Your Design

Managing different versions of design files in a large project can become difficult. To assist in this task, the Quartus II software provides utilities to copy and save different versions of your project. Creating a version of your project includes copying all your design files, your Quartus II settings file, and all your associated revisions (all assignments and settings).

To create a new version of your project with the Quartus II software, create a copy of your project and edit your design files. An example is if you have a design that is compatible with a 32-bit data bus and you require a new version of the design to interface with a 64-bit data bus. To solve this problem, create a new version of your project and edit the new version of the design files by performing the following steps:

1. On the Project menu, click **Copy Project.** The **Copy Project** dialog box appears (Figure 4–5).

Figure 4–5. Copy Project Dialog Box



2. Specify the path to your new project in the **Destination directory** box.
3. Type the new project name in the **New project name** box.
4. To open the new project immediately, turn on **Open new project**. This option closes the current project option.
5. Click **OK**.

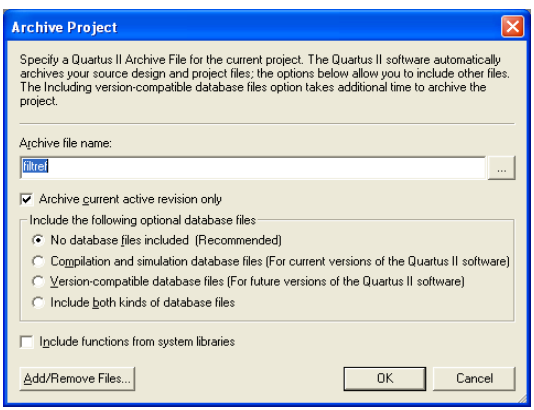
When creating a new version of your project with an Electronic Data Interchange Format (EDIF) or Verilog Quartus Mapping (.**vqm**) netlist from a third-party EDA synthesis tool, create a copy of your project and then replace the previous netlist file with the newly generated netlist file. On the Project menu, click **Copy Project** to create a copy of your design. On the Project menu, click the **Add/Remove Files** command to add and remove design files, if necessary.

Archiving Projects with the Quartus II Archive Project Feature

A single project can contain hundreds of files in many directories, which can make transferring a project between engineers difficult. You can use the Quartus II Archive Project feature to create a single compressed Quartus II Archive File (**.qar**) of your project containing all your design, project, and settings files. The Quartus II Archive File contains all the files, including the Quartus II Default Settings File (**.qdf**), required to perform a full compilation to restore the original results. The Quartus II Default Settings File contains all the project and assignment default settings from the current version of the Quartus II software. It is necessary to archive the Default Settings File to preserve your results when you restore the archive in a different version of the Quartus II software. For more information on the Quartus II Default Settings File, refer to "Quartus II Default Settings File" on page 4–26.

With the archive file generated by the Archive Project feature (Figure 4–6), you can easily share projects between engineers.

Figure 4–6. Archive Project Dialog Box



Archive a Project

To archive a project, perform these steps:

1. Create a new project or open an existing project. On the File menu, click **New Project Wizard** or **Open Project.**
2. On the Processing menu, point to Start and click **Start Analysis & Elaboration.**



Altera® recommends that you perform analysis and elaboration before archiving a project to ensure that all design files are located and archived.

3. On the Project menu, click **Archive Project**. The **Archive Project** dialog box appears (Figure 4–6).
4. Under **Archive file name**, type the file name of the Quartus II Archive File you want to archive, or click **Browse** to select a Quartus II Archive File name.
5. Turn on **Archive current active revision only** to archive the currently active revision. If you do not turn on this option, all revisions within the project are included in the project archive.
6. Under **Include the following optional database files**, select one of the following items:
    a. Select **No database files included** to exclude both compilation and simulation database files and version-compatible database files from the archive.
    b. Select **Compilation and simulation database files** to include the compilation and simulation database files in the archive.
    c. Select **Version-compatible database files** to include the version-compatible database files in the archive.
    d. Select **Include both kinds of database files** to include both compilation and simulation database files and version-compatible database files in the archive.
7. Turn on **Include functions from system libraries** to include functions from system libraries in the archive.
8. Click **Add/Remove Files** to add or remove files from the archive.
9. Click **OK**.

Restore an Archived Project

To restore an archived project, perform the following steps:

1. On the Project menu, click **Restore Archived Project**.
2. In the **Archive file name** box, type the file name of the Quartus II Archive File you wish to restore, or click **Browse to** select a Quartus II Archive File.
3. In the **Destination folder** box, specify the directory path into which you will restore the contents of the Quartus II Archive File, or browse to a directory.
4. Click **Show log** to view the Quartus II Archive Log File (**.qarlog**) for the project you are restoring from the Quartus II Archive File.
5. Click **OK**.
6. If you did not include the compilation and simulation database files in the project archive (Figure 4–6), you must recompile the project.

Version-Compatible Databases

Prior to the Quartus II software version 4.1, compilation databases were locked to the current version of the Quartus II software. With the introduction of the Version-Compatible Database feature in the Quartus II software version 4.1, you can export a version-compatible database and import it into a later version. For example, using one set of design files, you can export a database generated from the Quartus II software version 4.1 and import it into the Quartus II software version 5.1 and later without recompiling your design. Using this feature eliminates unnecessary compilation time.

Migrate to a New Version

To migrate a design from one Quartus II software version to a newer version, perform the following steps:

1. On the File menu, open the older version of the Quartus II software project by clicking **Open Project**.
2. On the Project menu, click **Copy Project** to make a new copy of the project. The copied project will open in the older version.
3. On the Project menu, click **Export Database**. By default, the database is exported to the **export_db directory** of the copied project. If desired, a new directory can be created.

4. Open the copied project from the new version of the Quartus II software. The Quartus II software deletes the existing database but not the exported database.

5. On the Project menu, click **Import Database**. By default, the directory of the database you just exported is selected.

Save the Database in a Version-Compatible Format

To save the database in a version-compatible format during every compilation, perform the following steps:

1. On the Assignments menu, click **Settings.** The **Settings** dialog box displays.

2. In the **Category** list, select the **Compilation Process** page. The Compilation Process page displays.

3. Turn on the **Export version-compatible database** option.

4. Browse to the directory where you want to save the database.

5. Click **OK**.

Quartus II Project Platform Migration

When moving your project from one computing platform to another, you must think about the following cross-platform issues:

- Filenames and Hierarchy
- Specifying Libraries
- Quartus II Search Path Precedence Rules
- Quartus II-Generated Files for Third-Party EDA Tools
- Migrating Database Files

Filenames and Hierarchy

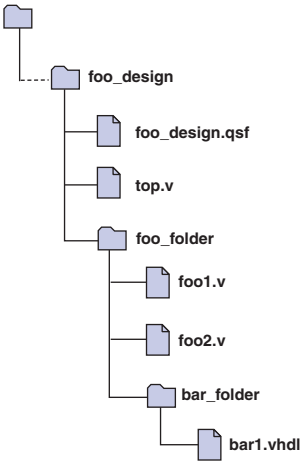
To ensure migration across platforms, you must consider a few basic differences between operating systems when naming source files, especially when interacting with these from the system-command prompt or a Tcl script:

- Some operating system file systems are case-sensitive. When writing scripts, ensure you specify paths exactly, even if the current operating system is not case-sensitive. For best results, use lowercase letters when naming files.
- Use a character set common to all the used platforms.

- You do not have to convert the forward-slash / and back-slash \ path separators in the Quartus Settings File because the Quartus II software changes all back-slash \ path separators to forward-slashes /.
- Observe the shortest file name length limit of the different operating systems you are using.

You can specify files and directories inside a Quartus II project as paths relative to the project directory. For instance, for a project titled **foo_design** with a directory structure shown in Figure 4–7, specify the source files as: **top.v, foo_folder/foo1.v**, **foo_folder/foo2.v**, and **foo_folder/bar_folder/bar1.vhdl**.

Figure 4–7. All-Inclusive Project Directory Structure



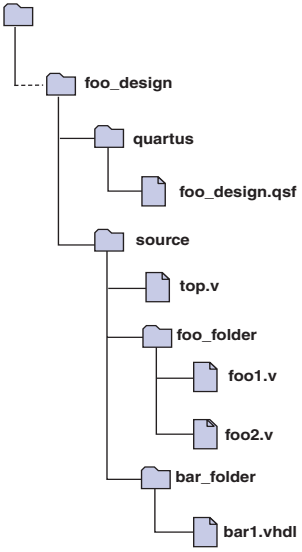
If the Quartus II Settings File is in a directory that is separate from the source files, you can specify paths using the following options:

- Relative paths
- Absolute paths
- Libraries

Relative Paths

If the source files are very near the Quartus II project directory, you can express relative paths using the . . notation. For example, given the directory structure shown in Figure 4–8, you can specify **top.v** as **../source/top.v** and **foo1.v** as **../source/foo_folder/foo1.v**.

Figure 4–8. Quartus II Project Directory Separate from Design Files



When you copy a directory structure to a different platform, ensure that any subdirectories are in the same hierarchical structure and relative path as in the original platform.

Specifying Libraries

You also can specify the directory (or directories) containing source files as a library that the Quartus II software searches when you compile your project. A Quartus II library is a directory containing design files used by your Quartus II project. You can specify the following two kinds of libraries in the Quartus II software:

- User libraries, which apply to a specific project
- Global libraries, which all projects use

Use the procedures in this section to specify user or global libraries.

All files in the directories specified as libraries are relative to the libraries. For example, if you want to include the file **/user_lib1/foo1.v** and the **user_lib1** directory is specified as a user library in the project, the **foo1.v** file can be specified in the Quartus II Settings File as **foo1.v**. The Quartus II software searches directories that are specified as libraries and finds the file.

Specifying User Libraries

To specify user libraries from the GUI: from the Assignments menu, click **Settings**, and select **User Libraries (Current Project).** Type the name of the directory in the **Library name** box, or browse to it. User libraries are stored in the Quartus II Settings File of the current revision.

Specifying Global Libraries

Specify global libraries from the GUI: On the Tools menu, click **Options**, and select **Global User Libraries (All Project).** Type the name of the directory in the **Library name** box, or browse to it. Global libraries are stored in the **quartus2.ini** file. The Quartus II software searches for the **quartus2.ini** file in the following order:

- USERPROFILE, for example, **C:\Documents and Settings**<*user name*>
- **Directory specified by the TMP** environmental variable
- **Directory specified by TEMP** environmental variable
- Root directory, for example, **C:**

For UNIX and Linux users, the file is created in the **altera.quartus** directory under the <*home*> directory, if the **altera.quartus** directory exists. If the **altera.quartus** directory does not exist, the file is created in the <*home*> directory.



Whenever you specify a directory name in the GUI or in Tcl, the name you use is maintained verbatim in the Quartus II Settings File rather than resolved to an absolute path.

If the directory is outside of the project directory, the path returned in the dialog box is an absolute path, and you can use the **Browse** button in either the **Settings** dialog box or the **Options** dialog box to select a directory. However, you can change this absolute path to a relative path by editing the absolute path displayed in the Library name field to create a relative path before you click **Add** to put the directory in the **Libraries** list.

When copying projects that specify user libraries, you must either copy your user library files along with the project directory or ensure that your user library files exist in the target platform.

Search Path Precedence Rules

If two files have the same file name, the file found is determined by the Quartus II software's search path precedence rules. The Quartus II software resolves relative paths by searching for the file in the following directories and order:

1. The project directory, which is the directory containing the Quartus II Settings File.
2. The project's database (**db)** directory.
3. User libraries are searched in the order specified by the **USER_LIBRARIES** setting of the Quartus II Settings File for the current revision.
4. Global user libraries are searched in the order specified by the **USER_LIBRARIES** setting on the **Global User Libraries** page in the **Options** dialog box.
5. The Quartus II software **libraries** directory.



For more information on libraries, refer to "Specifying Libraries Using Scripts" on page 4–29.

Quartus II-Generated Files for Third-Party EDA Tools

When you copy your project to another platform, regenerate any Quartus II software-generated files for use by other EDA tools, using the GUI or the **quartus_eda** executable.

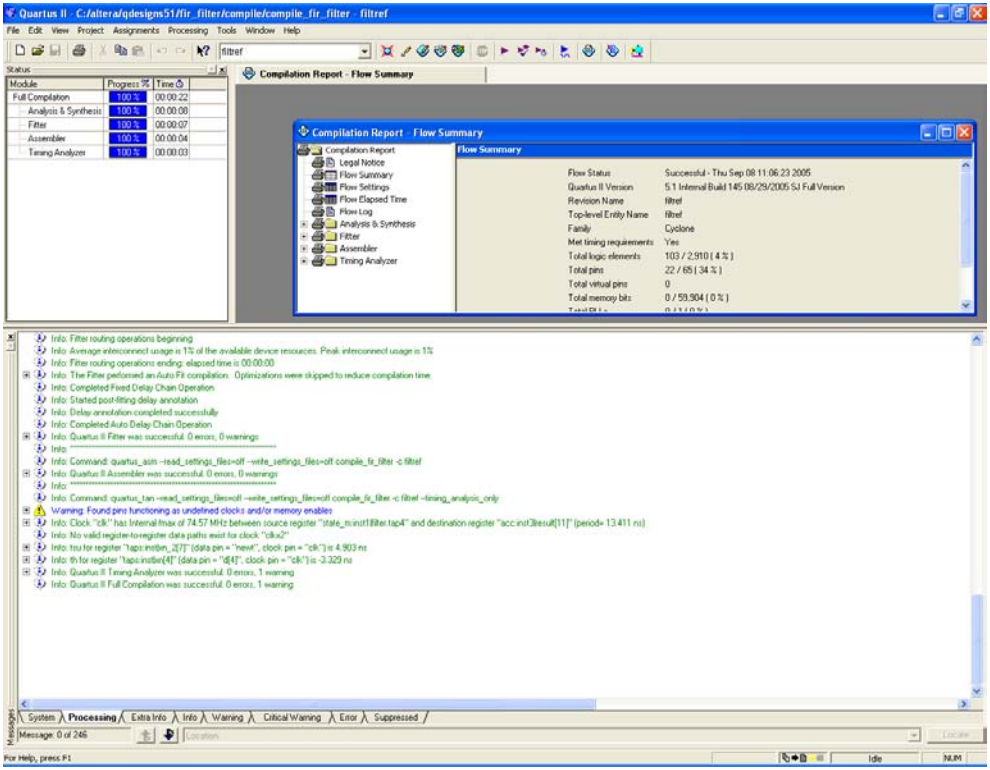
Migrating Database Files

There is nothing inherent in the file format and syntax of exported version-compatible database files that might cause problems for migrating the files to other platforms. However, the contents of the database can cause problems for platform migration. For example, use of absolute paths in version-compatible database files generated by the Quartus II software can cause problems for migration. Altera recommends that you change absolute paths to relative paths before migrating files whenever possible.

Working with Messages

The Quartus II software generates various types of messages, including Information, Warning, and Error messages. Some messages include information on software status during a compilation and alert you to possible problems with your design. Messages are displayed in the Messages window in the Quartus II GUI (Figure 4–9), and written to standard out and when you use command-line executables. In both cases, messages are written to Quartus II report files.

Figure 4–9. Viewing Quartus II Messages



You can right-click on a message in the Message window and get help on the message, locate the source of the message of your design, and manage messages.

Messages provide useful information if you take time to review them after each compilation. However, it can be tedious if there are thousands of them. Beginning with version 5.1 and later, the Quartus II software includes new features to help you manage messages.

Messages Window

By default, the Messages window displays eight message tabs (Table 4–1), which makes it easy to review all messages of a certain type.

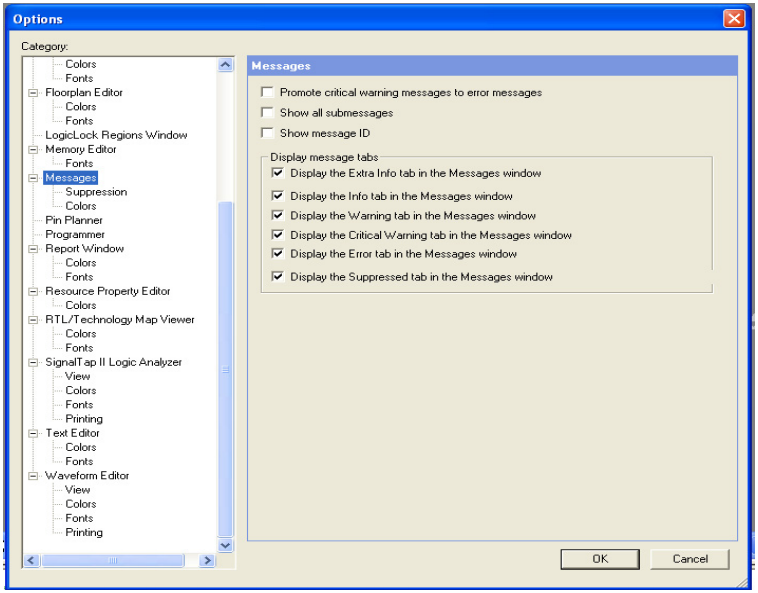
Table 4–1. Quartus II Message Tabs

Message Tab	Description
System	Displays messages that are unrelated to processing your design. For example, messages generated during programming are displayed in the **System** tab.
Processing	Displays messages that are generated when the Quartus II software processes your most recent compilation, simulation, or software build; timing analysis messages appear as part of the compilation messages.
Info	Displays general informational messages during a compilation, simulation, or software build. For example: legal and compilation-success messages.
Extra Info	Displays detailed informational messages about the operations for designers. For example: extra fitting information messages.
Warning	Displays strong warning messages generated during a compilation, simulation, or software build. For example: detection of signal promotion to global and high fan-out nets.
Critical Warning	Displays critical warning messages generated during a compilation, simulation, or software build. For example: detection of combinational feedback loops, gated clocks, or register duplication.
Error	Displays processing and compilation error messages generated during a compilation, simulation, or software build. Error messages can sometimes stop processing and cannot be disabled.
Suppressed	Displays suppressed messages during the last processing operation.

The **Info**, **Extra Info**, **Warning**, **Critical Warning**, and **Error** tabs display messages grouped by type. Warning messages are shown with all other types of messages in the Processing message window; all warning messages also appear in the **Warning message** tab.

You can control which tabs are displayed by right-clicking in the Messages window and choosing options from the right button pop-up menu, and with the options in the **Display Message** tabs section of the Messages page in the **Options** dialog box of the Tools menu (Figure 4–10).

Figure 4–10. Message Tab Options



The **Suppressed** tab shows messages suppressed during the last processing operation. You can also prevent the **Suppressed** tab from being displayed with an option in the **Display Message** tabs section of the Messages pane in the **Options** dialog box of the Tools menu.

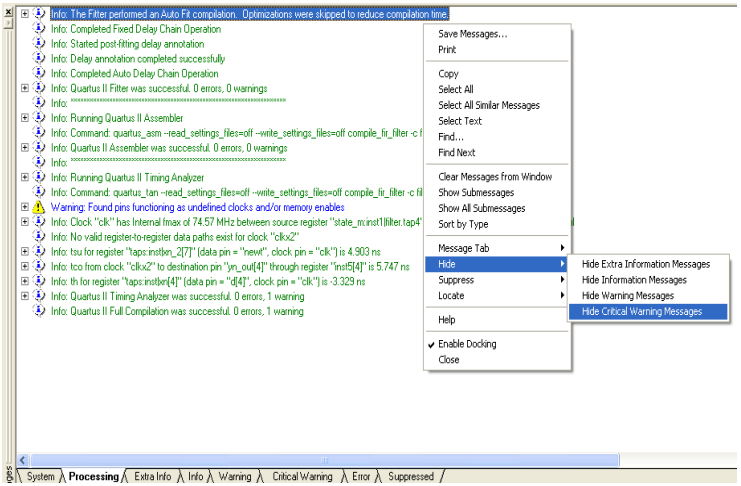
Hiding Messages

In the Messages window, you can hide all messages of a particular type. For example, to hide Info messages, follow these steps:

1. On the **Processing** tab, right-click in the Processing message window, and click the **Hide** option (Figure 4–11).

2. Select the Info message type.

Figure 4–11. Hiding Messages from the Processing Tab



All messages of the specified types are removed from the list of messages in the **Processing** tab, although they are still included in the separate tabs corresponding to the message type. For example, if you hide Info messages, no Info messages are shown in the Processing message window, but all the Info messages are shown in the Info messages window.

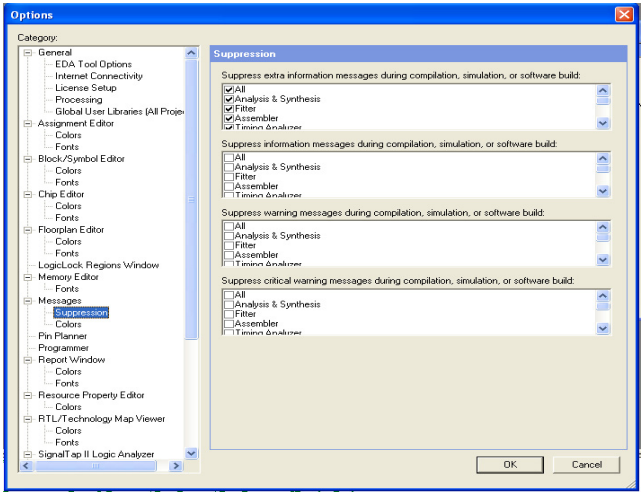
Message Suppression

Message suppression is a new feature in version 5.1 or later of the Quartus II software. You can use message suppression to reduce the number of messages to be reviewed after a compilation by preventing individual messages and entire categories of messages from being displayed. For example, if you review a particular message and determine that it is not caused by something in your design that should be changed or fixed, you can suppress the message so it is not displayed during subsequent compilations. This saves time because you see only new messages during subsequent compilations.

Every time you add a message to be suppressed, a suppression rule is created. Suppressing exact selected messages adds patterns that are exact strings to the suppression rules. Suppressing all similar messages adds patterns with wildcards to the suppression rules.

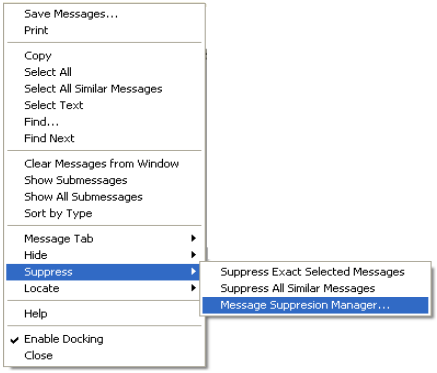
Furthermore, you can suppress all messages of a particular type in a particular stage of the compilation flow. On the Tools menu, click **Options, and click Suppression** from under the Messages section (Figure 4–12).

Figure 4–12. Controlling Suppression Messages



Suppressing individual messages is controlled in two locations in the Quartus II GUI. You can right-click on a message in the Messages window and choose commands in the Suppress sub-menu entry. To open the Message Suppression Manager, right-click in the Messages window. From the Suppress sub-menu item, click **Message Suppression Manager** (Figure 4–13).

Figure 4–13. Message Suppression Manager



Refer to "Message Suppression Manager" on page 4–22 for further information.

Message Suppression Methods

There are two methods you can use to create suppression rules: Suppress Exact Selected Messages and Suppress All Similar Messages. If you suppress a message with the exact selected messages option, only messages matching the exact text will be suppressed during subsequent compilations. The All Similar Messages option behaves like a wildcard pattern on variable fields in messages.

For an example of suppressing all similar messages, consider the following message:

```
Info: Found 1 design units, including 1 entities, in source file mult.v.
```

This type of message is common during synthesis and is displayed for each source file that is processed, with varying information about the number of design units, entities, and source file name.

Help for this message shows it is in the form Found <*number*> design units, including <*number*> entities, in source file <*name*>. Choosing to suppress all similar messages effectively replaces the variable parts of that message (<*number*>, <*number*>, and <*name*>) with wildcards, resulting in the following suppression rule:

```
Info: Found * design units, including * entities, in source file *.
```

As a result, all similar messages (ones that match the pattern) are suppressed.

Details and Limitations

The following limitations apply to which messages can be suppressed and how they can be suppressed:

- You cannot suppress error messages or messages with information about Altera legal agreements.
- Suppressing a message also suppresses all its submessages, if there are any.
- Suppressing a submessage causes matching submessages to be suppressed only if the parent messages are the same.
- You cannot create your own custom wildcards to suppress messages.
- You must use the Quartus II GUI to manage message suppression, including choosing messages to suppress. These messages are suppressed during compilation in the GUI and when using command-line executables.

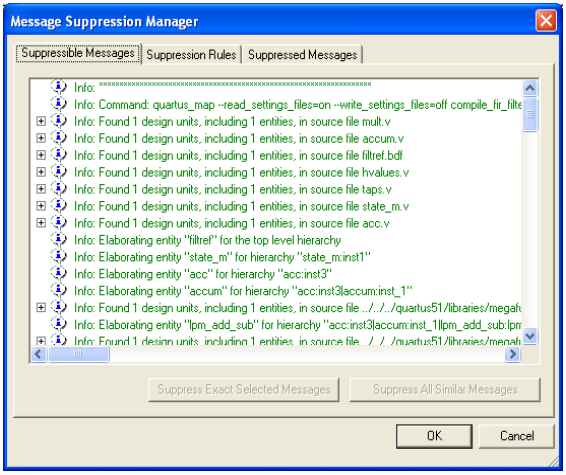
- Messages are suppressed on a per-revision basis, not for an entire project. Information about which messages to suppress is stored in a file called <*revision*>**.srf**. If you create a revision based on a revision for which messages are suppressed, the suppression rules file is copied to the new revision. You cannot make all revisions in one project use the same suppression rules file.
- You cannot remove messages or modify message suppression rules while a compilation is running.

Message Suppression Manager

You can use the Message Suppression Manager to view and suppress messages, view and delete suppression rules, and view suppressed messages.

Open the Message Suppression Manager by clicking the **Processing** tab. Right-click anywhere in the Messages window and click **Message Suppression Manager** from the Suppression sub-menu. The Message Suppression Manager has three tabs labeled **Suppressible Messages, Suppression Rules, and Suppressed Messages (**Figure 4–14).

Figure 4–14. Message Suppression Manager Window



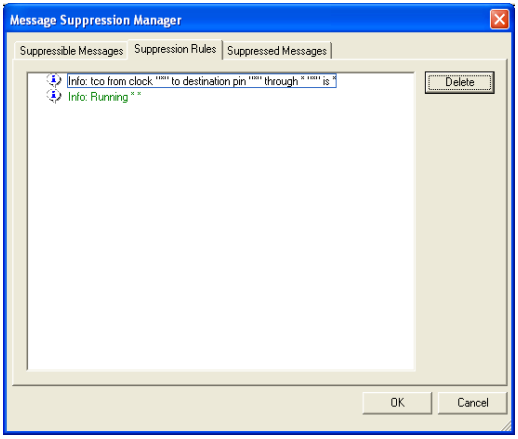
Suppressible Messages

Messages that are listed in the **Suppressible Messages** tab are messages that were not suppressed during the last compilation. These messages can be suppressed. The **Select All Similar Messages** option in the right click menu selects messages according to the example described in the "Message Suppression Methods" on page 4–21. You can select all similar messages to see which messages are suppressed if you choose to suppress all similar messages.

Suppression Rules

Items listed in the **Suppression Rules** tab are the patterns that the Quartus II software uses to determine whether to suppress a message. Messages matching any of the items listed in the **Suppression Rules** tab are suppressed during compilations (Figure 4–15).

Figure 4–15. Message Suppression Manager



An entry in the **Suppression Rules** tab that includes a message with submessages indicates the submessage is suppressed only when all its parent messages match.

You can stop suppressing messages by deleting the suppression rules that match them (causing them to be suppressed). Merely deleting suppression rules does not cause the formerly suppressed messages to be added to the messages generated during the previous compilation; you must recompile the design for the changed suppression rules to take effect.

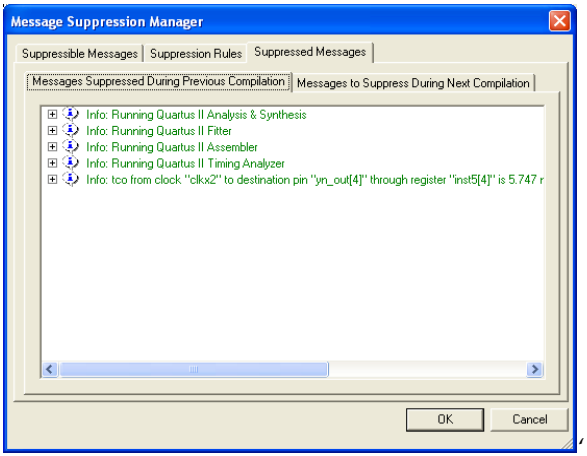
Suppressed Messages

Messages listed in the **Suppressed Messages** tab are divided in two sub-tabs:

- Messages Suppressed During Previous Compilation
- Messages to Suppress During Next Compilation

The messages listed in the Messages Suppressed During Previous Compilation sub-tab are all the suppressed messages from the previous compilation (Figure 4–16).

Figure 4–16. Messages Suppressed During Previous Compilation



These messages are also listed in the **Suppressed** tab in the Messages window. Messages listed in the Messages to Suppress During Next Compilation are messages that will be suppressed during the next compilation that match suppression rules created after the last compilation finished.

In addition to appearing in the **Suppressed** tab in the Messages window, suppressed messages are included in a Suppressed Messages entry in the Quartus II compilation report, viewable in the GUI. Suppressed messages are not included in the *<revision>*.*<module>*.**rpt** text files; they are written to a separate text report file called *<revision name>*.*<module>*.**smsg**.

Quartus II Settings File

All assignments made in the Quartus II software are stored as Tcl commands in the Quartus II Settings File. The Quartus II Settings File is a text based file containing Tcl commands and comments. The Quartus II Settings File is not a Tcl script and does not support the full Tcl scripting language.

As you make assignments in the Quartus II software, the assignments are either stored temporarily in memory or written out to the Quartus II Settings File. This is determined by the **Update assignments to disk during design processing only** option, which is located in the Tools menu under Options on the Processing page. If the option is turned on, then all assignments are stored in memory and are written to the Quartus II Settings File when a compilation has started or when you save or close the project. By saving assignments to memory, the performance of the software is improved because it avoids unnecessary reading and writing to the Quartus II Settings File on the disk. This performance improvement is seen more dramatically when the project files are stored on a remote data disk.

Beginning with the Quartus II software version 5.1, you can add lines of comments into the Quartus II Settings File, such as are shown in the following example:

```
# Assignments for input pin clk
# Clk is being driven by FPGA 1
set_location_assignment PIN_6 -to clk
set_instance_assignment -name IO_STANDARD "2.5 V" -to clk
```

Sourcing other Quartus II Settings Files is supported using the following Tcl command:

```
source <filename>.qsf
```

Format Preservation

Beginning with the Quartus II software version 5.1, the Quartus II software maintains the order of assignments within the Quartus II Settings File. When you make new assignments, they are appended to the end of the Quartus II Settings File. If you modify an assignment, the corresponding line in the Quartus II Settings File is modified and the order of assignments in the Quartus II Settings File is maintained except when you add and remove project source files, or when you add, remove, and exclude members from an assignment group. In these cases, all assignments are moved to the end of the Quartus II Settings File. For example, if you add a new design file into the project, the list of all your design files is removed from its current location in the file and moved to the end of the Quartus II Settings File.



The header that is located at the beginning of the Quartus II Settings File is written only if the Quartus II Settings File is newly created.

The Quartus II software preserves all spaces and tabs for all unmodified assignments and comments. When you make a new assignment or modify an existing assignment, the assignment is written using the default formatting.

Quartus II Default Settings File

The Quartus II Default Settings File contains all the project and assignment default settings from the current version of the Quartus II software. The Quartus II Default Settings File, located in the win directory of the Quartus II installation path, is used to ensure consistent results when defaults are changed between versions of the Quartus II software.

The Quartus II software reads assignments from various files and stores the assignments in memory. The Quartus II software reads settings files in the following order shown below, so that assignments in subsequent files take precedence over earlier ones:

1. **assignment_defaults.qdf** from <*Quartus II Installation directory*>**/win**
2. **assignment_defaults.qdf** from project directory
3. <*revision name*>**_assignment_defaults.qdf** from project directory
4. <*revision name*>**.qsf** from project directory

As each new file is read, if an existing assignment from a previous file matches (following rules of case sensitivity, multi-value fields as well as other rules), then the old value is removed and replaced by the new. For example, if the first file has a non multi-valued assignment A=1, and the second file has A=2, then the assignment A=1, stored in memory, is replaced by A=2.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```

The *Scripting Reference Manual* includes the same information in PDF form.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Managing Revisions

You can use the following commands to create and manage revisions. For more information about managing revisions, including creating and deleting revisions, setting the current revision, and getting a list of revisions, refer to "Creating and Deleting Revisions" on page 4–3.

Creating Revisions

The following Tcl command creates a new revision called **speed_ch**, based on a revision called **chiptrip** and sets the new revision as the current revision. The **–based_on** and **–set_current** options are optional.

```
create_revision speed_ch -based_on chiptrip -set_current
```

Setting the Current Revision

Use the following Tcl command to specify the current revision:

```
set_current_revision <revision name>
```

Getting a List of Revisions

Use the following Tcl command to get a list of revisions in the opened project:

```
get_project_revisions
```

Deleting Revisions

Use the following Tcl command to delete a revision:

```
delete_revision <revision name>
```

Archiving Projects with a Tcl Command or at the Command Prompt

You can archive projects with a Tcl command or with a command run at the system command prompt.

The following Tcl command creates a project archive with the default settings and overwrites the specified archived file if it already exists:

```
project_archive archive.qar -overwrite
```

Type the following command at a command prompt to create a project archive called **top**:

```
quartus_sh --archive top ↵
```

Restoring Archived Projects

You can restore archived projects with a Tcl command or with a command run at a command prompt. For more information about restoring archived projects, refer to "Restore an Archived Project" on page 4–10.

The following Tcl command restores the project archive named **archive.qar** in the **restored** subdirectory and overwrites existing files:

```
project_restore archive.qar -destination restored -overwrite
```

Type the following command at a command prompt to restore a project archive:

```
quartus_sh --restore archive.qar ↵
```

Importing and Exporting Version-Compatible Databases

You can import and export version-compatible databases with either a Tcl command or a command run at a command prompt. For more information about importing and exporting version-compatible databases, refer to "Version-Compatible Databases" on page 4–10.



The `flow` and `database_manager` packages contain commands to manage version-compatible databases.

Use the following Tcl commands from the `database_manager` package to import or export version-compatible databases.

```
export_database <directory>
import_database <directory>
```

Use the following Tcl commands from the `flow` package to import or export version-compatible databases. If you use the flow package, you must specify the database directory variable name.

```
set_global_assignment \
-name VER_COMPATIBLE_DB_DIR <directory>
execute_flow –flow export_database
execute_flow –flow import_database
```

Add the following Tcl commands to automatically generate version-compatible databases after every compilation:

```
set_global_assignment \
-name AUTO_EXPORT_VER_COMPATIBLE_DB ON
set_global_assignment \
-name VER_COMPATIBLE_DB_DIR <directory>
```

The **quartus_cdb** and the **quartus_sh** executables provide commands to manage version-compatible databases:

```
quartus_cdb <project> -c <revision> \
--export_database=<directory> ↵
quartus_cdb <project> -c <revision> \
--import_database=<directory>↵

quartus_sh –flow export_database <project> -c \
<revision> ↵
quartus_sh –flow import_database <project> -c \
<revision> ↵
```

Specifying Libraries Using Scripts

In Tcl, use commands in the `::quartus::project` package to specify user libraries. To specify user libraries, use the **set_global_assignment** command. To specify global libraries use the **set_user_option** command. The following examples show typical usage of the **set_global_assignment** and **set_user_option** commands:

```
set_global_assignment -name USER_LIBRARIES \
"../other_dir/library1"
set_user_option -name USER_LIBRARIES \
"../an_other_dir/library2"
```

To report any user libraries specified for a project and any global libraries specified for the current installation of the Quartus II software, use the **get_global_assignment** and **get_user_option** Tcl commands. The following Tcl script outputs the user paths and global libraries for an open Quartus II project:

```
get_global_assignment -name USER_LIBRARIES
get_user_option -name USER_LIBRARIES
```

Conclusion

Designers often try different settings and versions of their designs throughout the development process. Quartus II project revisions facilitate the creation and management of different assignments and settings.

In addition, understanding how to smoothly migrate your projects from one computing platform to another, controlling messages, and reducing compilation time is important as well. The Quartus II software facilitates efficient management of your design to accommodate today's more sophisticated FPGA designs.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 4–2 shows the revision history for this chapter.

Table 4–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 4–30.	—
May 2007 v7.1.0	No changes made to content.	—
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	No changes made.	—
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.0.	—
May 2005 v5.0.0	Updated for the Quartus II software version 5.0.0.	—
December 2004 v1.1	Updated for Quartus II software version 4.2: • General formatting and editing updates. • Added new figures. • Added new introduction to To Delete a Revision That is a Design's Current Revision. • Added new section To Delete a Revision That is not a Design's Current Revision. • Updated figures. • Added new information about displaying assignments for multiple revisions. • Updated Archive a Project. • Updated Restore an Archived Project. • Version-Compatible Databases describes migration to Quartus II software version 4.2. • Corrected Tcl commands.	—
June 2004 v1.0	Initial release.	—

Section II. I/O and PCB Tools

This section provides an overview of the I/O planning process, Altera's FPGA pin terminology, as well as the various methods for importing, exporting, creating, and validating pin-related assignments using Quartus® II software. This section also describes the design flow that includes making and analyzing pin assignments using the **Start I/O Assignment Analysis** command in the Quartus II software, during and after the development of your HDL design. It also describes interfaces with third-party PCB design tools

This section includes the following chapters:

- Chapter 5, I/O Management
- Chapter 6, Mentor Graphics PCB Design Tools Support
- Chapter 7, Cadence PCB Design Tools Support



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

5. I/O Management

QII52013-7.2.0

Introduction

The process of managing I/Os for current leading FPGA devices involves more than just fitting design pins into a package. The increasing complexity of I/O standards and pin placement guidelines are just some of the factors that influence pin-related assignments. The I/O capabilities of the FPGA device and board layout guidelines influence pin location and other types of assignments for each of your design pins. Therefore, it is necessary to begin I/O planning and printed circuit board (PCB) development even before starting the FPGA design.

This chapter provides an overview of the I/O planning process, FPGA pin terminology and the various methods for importing, exporting, creating, and validating pin-related assignments.



For guidelines on PCB designs for Altera® high-speed FPGAs, refer to *AN 315: Guidelines for Designing High-Speed FPGA PCBs*.

This chapter contains the following topics:

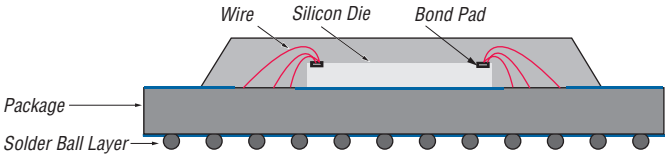
- "Understanding Altera FPGA Pin Terminology" on page 5–2
- "Importing and Exporting Pin Assignments" on page 5–6
- "I/O Planning Overview" on page 5–10
- "Early I/O Planning Using the Pin Planner" on page 5–13
- "Creating Pin-Related Assignments" on page 5–21
- "Using the Live I/O Check Feature to Validate Pin Assignments" on page 5–65
- "Using I/O Assignment Analysis to Validate Pin Assignments" on page 5–67
- "Incorporating PCB Design Tools" on page 5–87
- "Advanced I/O Timing" on page 5–87

Understanding Altera FPGA Pin Terminology

Altera FPGA devices are available in a variety of packages to meet all of your complex design needs. To describe Altera FPGA pin terminology, this chapter uses a wirebond ball grid array (BGA) package in its examples. On the top surface of the silicon die, there is a ring of bond pads that connect to the I/O pins of the silicon. In a wirebond BGA package, the device is placed in the package and copper wires connect the bond pads to the solder balls of the package. Figure 5–1 shows a cross section of a wirebond BGA package.

For a list of all BGA packages available for each Altera FPGA device, refer to the *Altera Device Package Information Datasheet*.

Figure 5–1. Wire Bond BGA

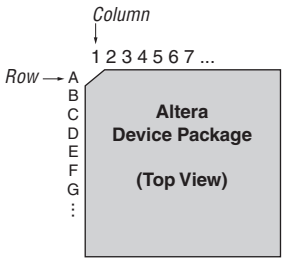


Package Pins

The pins of a BGA package are small solder balls arranged in a grid pattern on the bottom of the package. In the Quartus® II software, the package pins are represented as pin numbers. The pin numbers are determined by their locations using a coordinate system with letters and numbers identifying the row and column of the pins, respectively.

The upper-most row of pins is labeled "A" and continues alphabetically as you move downward (Figure 5–2). The left-most column of pins is labeled "1" and continues with increments of 1 as you move to the right. For example, pin number "B4" represents row "B" and column "4."

Figure 5–2. Row and Column Labeling



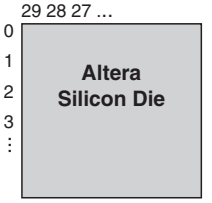
The letters I, O, Q, S, X, and Z are never used in pin numbers. If there are more rows than letters of the alphabet, then the alphabet is repeated, prefixed with the letter "A."

For more information about the pin numbers for your Altera device, refer to the device pin-out information available on the Altera website at **www.altera.com**.

Pads

Package pins are connected to pads located on the perimeter of the top metal layer of the silicon die (Figure 5–1). Each pad is identified by a pad ID, which is numbered starting at 0, incrementing by 1 in a counterclockwise direction (Figure 5–3).

Figure 5–3. Pad Number Ordering



To prevent signal integrity issues, the Quartus II software uses pin placement rules to validate your pin placements and pin-related assignments. It is important that you understand which pad locations your pins were assigned, because some pin placement rules describe pad placement restrictions. For example, in certain devices, there is a restriction on the number of I/O pins supported by a VREF pad to ensure signal integrity. There are also restrictions on the number of pads between single-ended input or output pins and a differential pin. The Quartus II software performs pin placement analysis, and if pins are not placed according to pin placement rules, the design compilation fails and the Quartus II software reports an error.

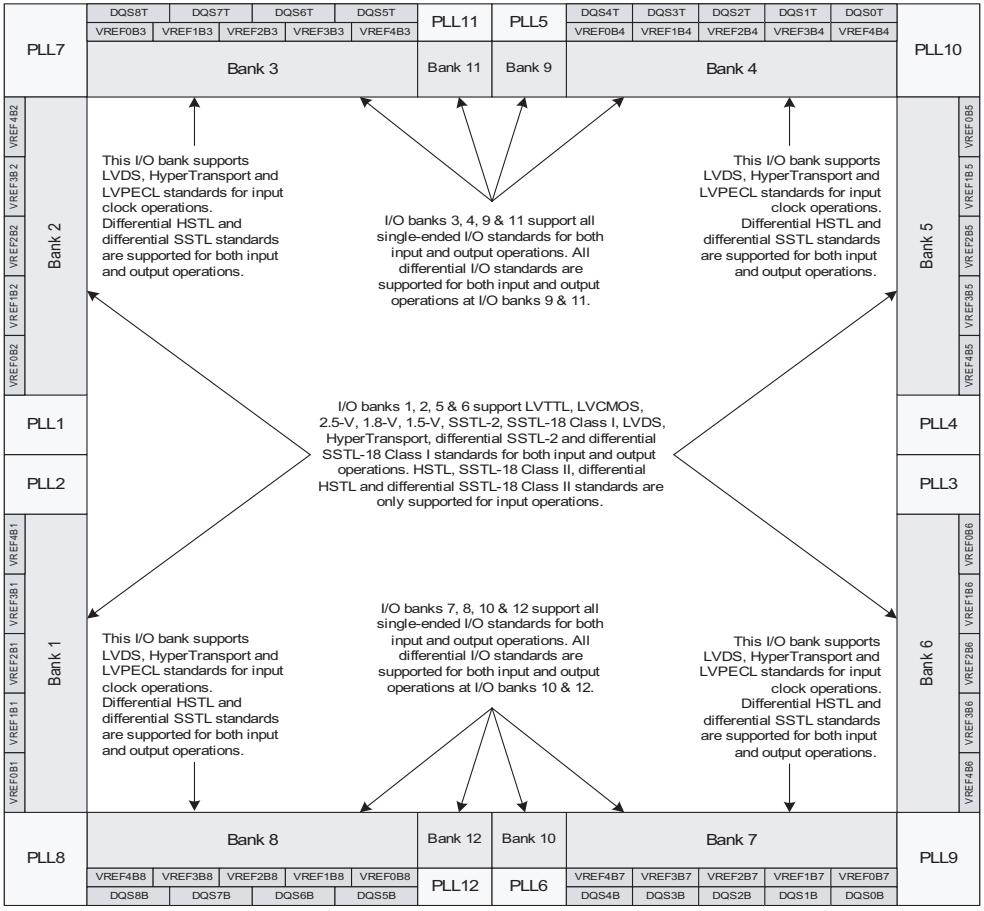
For more information about pin placement guidelines, refer to the *Selectable I/O Standards* chapter in volume 1 of the appropriate device handbook.

I/O Banks

I/O pins are organized into I/O banks designed to facilitate the various supported I/O standards. Each I/O bank is numbered and has its own voltage source pins, called VCCIO, to offer the highest I/O performance. Depending on the device and I/O standards for the pins within the I/O bank, the specified voltage of the VCCIO pin is between 1.5 V and 3.3 V. Each I/O bank can support multiple pins with different I/O standards that share the same V_{CCIO}.

It is important to refer to the appropriate device handbook to determine the capabilities of each I/O bank. For example, the pins in the I/O banks on the left and right side of a Stratix® II device support high-speed I/O standards such as LVDS, whereas the pins on the top and bottom I/O banks support all single-ended I/O standards, including data strobe signaling (DQS) (Figure 5–4). Pins belonging to the same I/O bank must use the same VCCIO signal.

Figure 5–4. Stratix II I/O Banks *Notes (1), (2), (3), (4)*



Notes to Figure 5–4:

(1) This figure shows a top view of the silicon die which corresponds to a reverse view for flip chip packages. It is a graphical representation only.

(2) Depending on the size of the device, different device members have a different number of V_{REF} groups. Refer to the pin list and the Quartus II software for exact locations.

(3) Banks 9 through 12 are enhanced phase locked loop (PLL) external clock output banks.

(4) Horizontal I/O banks feature SERDES and DPA circuitry for high speed differential I/O standards. For more information about differential I/O standards, refer to the *High-Speed Differential I/O Interfaces with DPA in Stratix II and Stratix II GX Devices* chapter in volume 2 of the *Stratix II Device Handbook*.

VREF Groups

A VREF group is a group of pins that includes one dedicated VREF pin as required by voltage-referenced I/O standards. A VREF group is made up of a small number of pins, as compared to the I/O bank, to maintain the signal integrity of the VREF pin. One or more VREF groups exist in an I/O bank. The pins in a VREF group share the same V_{CCIO} and V_{REF} voltages.



For more information about I/O banks, VREF groups, and supported I/O standards, refer to the *Architecture* and *Selectable I/O Standards* chapters in the appropriate device handbook.

Importing and Exporting Pin Assignments

You can transfer pin-related assignments between the Quartus II software and other tools by importing and exporting these assignments in the following file formats: Comma Separated Value (**.csv**) file, Quartus II Settings File (**.qsf**), Tool command language (Tcl), FPGA Xchange (**.fx**) file, and Pin-Out (**.pin**) file (export only).

CSV File

You can transfer pin-related assignments as a CSV file. This file consists of a row of column headings followed by rows of comma-separated data. The row of column headings in the exported file is in the same order and format as the columns displayed in the Pin category in the Assignment Editor or in the All Pins List in the Pin Planner when the export is performed. Do not modify the row of column headings if you plan to import the CSV file later.

To import a CSV file into your project, on the Assignment menu, click **Import Assignments** and browse to the file.

You can export pin-related assignments from the Quartus II Pin Planner or the Assignment Editor. To export your pin-related assignments to a CSV file, on the Assignment menu, click **Pin Planner** or **Assignment Editor**. For the Pin Planner, make sure the All Pins List is visible. If the list is not visible, on the View menu, click **All Pins List**. For the Assignment Editor, from the **Category** list, select the **Pin** category. Then, to create the CSV file, on the File menu, click **Export**.



The All Pins List in the Quartus II Pin Planner, the Pin category in the Quartus II Assignment Editor, and the device PIN files all display detailed properties about each pin of the device, in addition to the pin name and pin number. The device PIN files are available on the Altera web site at **www.altera.com**.



For more information about importing and exporting CSV files and the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

Quartus II Settings Files (QSFs)

You can transfer pin-related assignments as a QSF. The pin-related assignments are stored as Tcl commands in the QSF.

To import a QSF, on the Assignments menu, click **Import Assignments** and browse to the file, or source the file in the Tcl console. To export a QSF, on the Assignments menu, click **Export Assignments**, type in a file name, and click **OK**.



For more information about QSFs, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

Tcl Script

To import the pin-related assignments from a Tcl script, source the Tcl script in the Tcl console or run the Tcl script with the `quartus_sh` executable. For example, type the following command at a system command prompt:

```
quartus_sh -t my_pins.tcl ↵
```

You can export pin-related assignments from the Quartus II Pin Planner or the Assignment Editor. To export pin-related assignments as a Tcl script, on the Assignments menu, click **Pin Planner** or **Assignment Editor**. For the Pin Planner, make sure the All Pins List is visible. If the list is not visible, on the View menu, click **All Pins List**. For the Assignment Editor, select the **Pin** category from the **Category** list. Then, to create the Tcl file, on the File menu, click **Export**. In the **Export** dialog box, type in a file name, select **Tcl Script File** (***.tcl**), and click **OK**. All pin-related assignments displayed in the All Pins List of the Pin Planner and the spreadsheet of the Assignment Editor are saved as Tcl commands in the Tcl script.

For more information about the All Pins List in the Pin Planner, refer to "Using the Pin Planner" on page 5–22.



For more information about Quartus II scripting support including examples, refer to the *Tcl Scripting* and *Command-Line Scripting* chapters in volume 2 of the *Quartus II Handbook*.

FPGA Xchange File

An FPGA Xchange file contains device and pin-related information that allows you to transfer information between the Quartus II software and your PCB schematic or design tool. For example, to transfer pin information from the Mentor Graphics I/O Designer software to the Quartus II software to validate pin assignment changes using the I/O Assignment Analyzer, use an FPGA Xchange file.

To import an FPGA Xchange file into the Quartus II software, perform the following steps:

1. On the Assignments menu, click **Import Assignments**.
2. In the **File name** box, click the browse button and click FPGA Xchange Files **(*.fx)** in the **Files of type** list.
3. Browse to and select the FPGA Xchange file and click **Open**.
4. Click **OK**.

To generate an FPGA Xchange file in the Quartus II software, perform the following steps:

1. Perform an I/O Assignment Analysis or a full compilation in the Quartus II software.
2. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
3. Select **Board-Level**. The **Board-Level** page appears.
4. Under **Board-level symbol**, in the **Format** list, select **FPGA Xchange**.
5. Set the Output directory to the location where you want to save the file. The default output file path is *<project directory>***/symbols/fpgaxchange**.
6. Click **OK**.
7. On the Processing menu, point to Start and click **Start EDA Netlist Writer**.

The output directory you selected is created when you generate the FPGA Xchange file using the Quartus II software.

PIN File

A PIN file is an ASCII text file containing pin location results and other pin information. To generate a PIN file for your project, you must successfully perform an I/O Assignment Analysis or full compilation.

Use the PIN file to understand which signals should be connected to which pins, or to transfer your project's pin information into third-party PCB tools for board development. Figure 5–5 shows an example PIN file, and Table 5–1 describes the columns in a PIN file.

Figure 5–5. Example of a PIN File

Pin Name/Usage	Location	Dir.	I/O Standard	Voltage	I/O Bank	User Assignment
VCCA_PLL1 clk	9 10	power input	 LVTTL	1.5V	 1	 N

Table 5–1. PIN File Header Description

Column Name	Description
Pin Name/Usage	The name of a design pin, ground, or power
Location	The pin number of the location on the device package
Dir	The direction of the pin
I/O Standard	The name of the I/O standard to which the pin is configured
Voltage	The voltage level that is required to be connected to this pin
I/O Bank	The I/O bank number that the pin belongs to
User Assignment	Y or N indicating if the location assignment for the design pin was user assigned (Y) or assigned by the Fitter (N)

For more information about Pin Name/Usage, refer to the Device Pin-Out for the targeted device, available on the Altera website at **www.altera.com**.

For more information about using Cadence PCB tools with the Quartus II software, refer to the *Cadence PCB Design Tools Support* chapter in volume 2 of the *Quartus II Handbook*. For more information about using the Mentor Graphics PCB tools with the Quartus II software, refer to the *Mentor Graphics PCB Design Tools Support* chapter in volume 2 of the *Quartus II Handbook*.

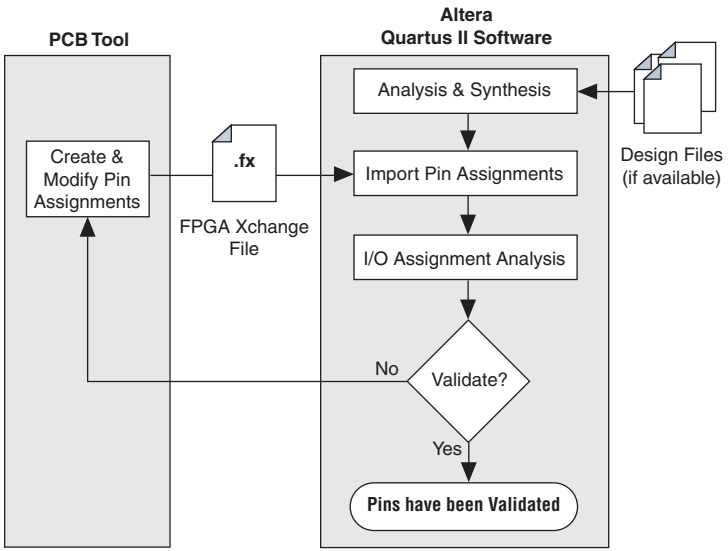
I/O Planning Overview

I/O planning includes importing any existing pin assignments, optionally using the early pin planning flow, creating and editing pin-related assignments and validating them against pin placement rules. The I/O planning process ensures a successful fit in your Altera FPGA device. The Quartus® II software includes the Pin Planner and I/O Assignment Analysis to assist you in I/O planning.

The method you use to create your pin assignments depends on your requirements. If your PCB is partially designed, create your FPGA assignments in your PCB tool and import them into the Quartus II software for validation (Figure 5–6).

Currently, only the Mentor Graphics® I/O Designer PCB tool is supported in this reverse I/O planning flow.

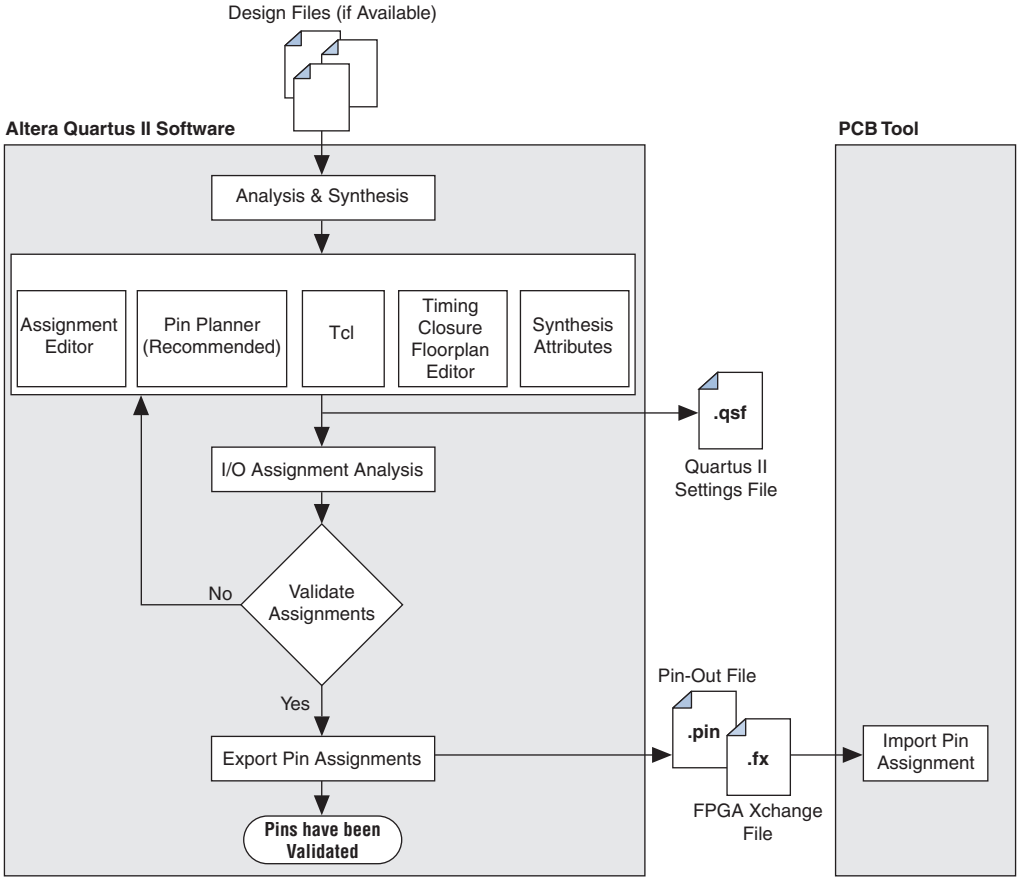
Figure 5–6. I/O Planning Flow Using an FPGA Xchange File from a PCB Tool



For more information about board layout and I/O pin assignment import and export, refer to the *Cadence PCB Design Tools Support* and the *Mentor Graphics PCB Design Tools Support* chapters in volume 2 of the *Quartus II Handbook*.

If you have not designed the PCB yet, create and validate your I/O assignments in the Quartus II software, then export them to the PCB tool (Figure 5–7). This is the recommended design flow for creating I/O assignments for an FPGA design.

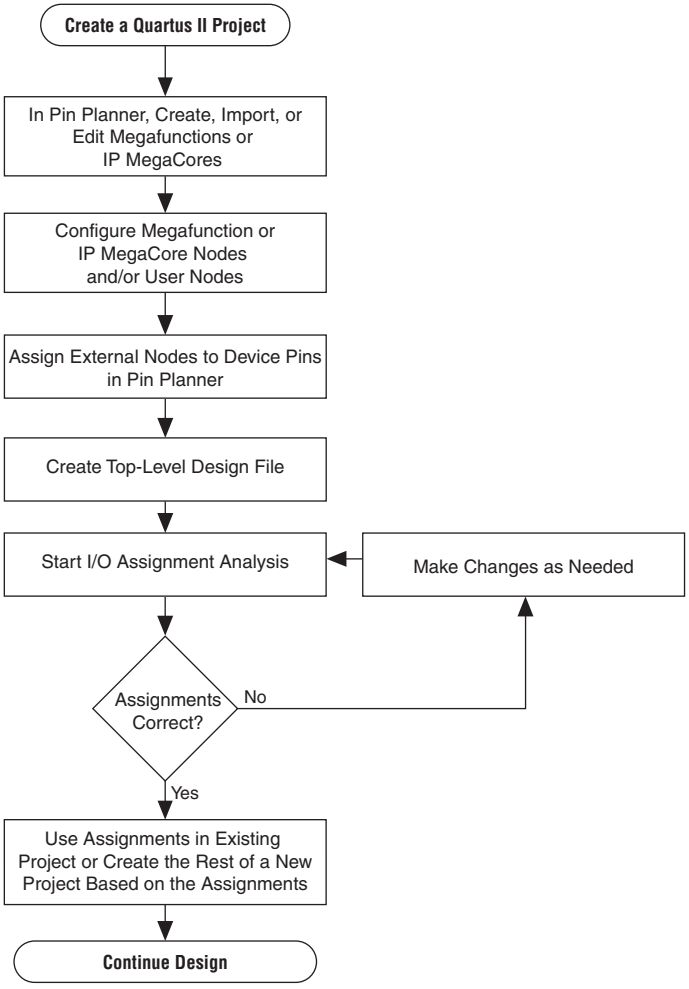
Figure 5–7. Quartus II Software I/O Planning Flow



The preferred method for validating pin-related assignments is to perform a full compilation. If design files are not available, create a top-level netlist wrapper file while making pin assignments and creating custom megafunctions (Figure 5–8). With the wrapper file, you can use the I/O Assignment Analysis to validate your I/O assignments early in the FPGA design process.

For more details about this early I/O planning design flow, refer to "Using I/O Assignment Analysis to Validate Pin Assignments" on page 5–67.

Figure 5–8. Early I/O Planning Using the Pin Planner



Early I/O Planning Using the Pin Planner

It may be difficult to plan your I/Os early in the design cycle because the design files, including the top-level design, may not be available yet. However, the interfaces between your FPGA and other devices are typically determined and documented in the design specifications. By adding the bus or memory interfaces needed to connect your FPGA with these other devices in the Pin Planner, you can plan your FPGA I/Os efficiently without design files.

The Pin Planner can interface with the MegaWizard® Plug-in Manager, allowing you to create or import custom megafunctions and intellectual-property (IP) cores. You can add many types of interfaces, including megafunctions such as altpll and altddio, and IP MegaCores such as PCI Compiler, QDR II, and Rapid IO. Adding the interface information while planning your I/Os allows you to assign each required pin without manually creating each pin individually in the Pin Planner. Furthermore, you can add and configure your own user ports that are top-level ports in your design.



Any IP megafunction that has a pin planning file (**.ppf**) can be imported to the Pin Planner. For details about creating a specific megafunction, refer to the user guide for that megafunction on the User Guides literature page of the Altera website at **www.altera.com**.

Reading in the pin planning files and defining the top-level ports automatically populates the All Pins list and the Groups list in the Pin Planner with all the external pins of your megafunctions and IP MegaCores. For more information about the All Pins list and the Groups list, refer to "Using the Pin Planner" on page 5–22.

You can then make I/O pin assignments for all the external pins of your interfaces, create the required top-level wrapper file, and validate the assignments. Figure 5–8 on page 5–12 shows a flow diagram for this type of early I/O planning flow.

After you complete and validate the I/O assignments, you can proceed with your design in any of the following ways:

- Transfer the assignments to an existing project that includes design files, making sure the pin names match the design.
- Continue working with this early I/O project, adding design files to work with the planned I/O assignments.
- Create a revision of your existing design that uses the wrapper file and verified I/O assignments and decide later whether to integrate them with your project.



For information about revisions in the Quartus II software, refer to the Quartus II Help.

Create a Megafunction or IP MegaCore Variation from the Pin Planner

You can create or customize some megafunction variations from within the Pin Planner. To create a megafunction or IP MegaCore variation from the Pin Planner, perform the following steps:

1. In the Pin Planner, right-click anywhere in the Groups List or All Pins List.
2. On the shortcut (right-click) menu, click **Create/Import Megafunction**. The **Create/Import Megafunction** dialog box appears. You can also open this dialog box from the Edit menu or directly from the Toolbar.
3. To create a new megafunction, select **Create a new custom megafunction** and click **OK**. The **MegaWizard Plug-In Manager** dialog box appears.
4. In the list of all of the supported megafunctions and IP MegaCores, select the megafunction or IP MegaCore you want to create, and complete the MegaWizard Plug-In pages.
5. After you complete the wizard, a new group, based on the file name you provided, is created and all the I/O names, directions, and I/O standards are listed as members of the group in the Groups List. Make pin location assignments for the group or for each individual pin.



For more information about a particular megafunction, refer to the appropriate megafunction user guide, available on the User Guides literature page of the Altera website at **www.altera.com**.

Import a Megafunction or IP MegaCore Variation from the Pin Planner

To import a megafunction variation to the Pin Planner, perform the following steps:

1. In the Pin Planner, right-click anywhere in the Groups List or All Pins List.

2. On the right-click menu, click **Create/Import Megafunction**. The **Create/Import Megafunction** dialog box appears. You can also open this dialog box from the Edit menu.

3. Select **Import an existing custom megafunction** and click the browse button**.** Select the Pin Planner File (**.ppf**) that was generated along with your megafunction variation or your IP MegaCore files.

4. In the **Instance name** box, type in an instance name and click **OK**.



To avoid pin name conflicts when there is more than one instance of a megafunction or IP MegaCore, the instance name is appended to the beginning of each pin name.

When the wizard is complete, a new group based on the file name you provided is created, and all the I/Os that are used externally are listed as members of the group. Make pin location assignments for the group or for each individual pin.

Create a Top-Level Design File for I/O Analysis

You can create a top-level design file after you add or modify user ports, megafunction nodes, or IP MegaCore nodes in your project with the Pin Planner. Even before the internal logic is defined, the top-level design file enables you to validate your I/O assignments and provides a base on which to build the rest of your design. Before you create a top-level design file, you must first configure the user ports, megafunction nodes, and IP MegaCore nodes created in the Pin Planner for integration with each other and the rest of the design.

Configure Megafunction Nodes

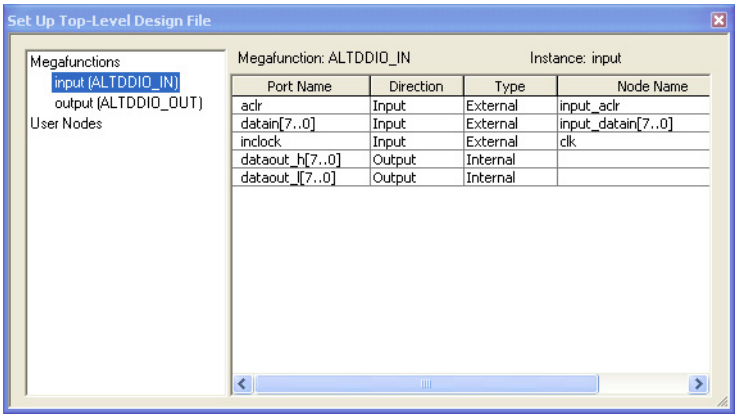
After creating or importing custom megafunctions or IP MegaCores in the Pin Planner, you must configure how they will be connected to each other. You do this by specifying matching node names for selected ports of the megafunctions or IP MegaCores.



In this section, ports and port names refer to the generic port names of megafunctions and IP MegaCores in the MegaWizard Plug-In Manager. Node names refer to the unique names assigned to ports when the megafunction or IP MegaCore is created based on the instance name given when the MegaWizard Plug-In Manager is started. By default, node names are the original port names prefixed with *<instance name>*_.

To configure your custom megafunctions and IP MegaCores for creating a top-level design file, on the Edit menu of the Package View, click **Set Up Top-Level Design File**. The **Set Up Top-Level Design File** dialog box appears (Figure 5–9).

Figure 5–9. Set Up Top-Level Design File Dialog Box



Click the name of a megafunction or IP MegaCore in the list on the left. The list on the right contains all the ports for the selected megafunction.

The columns in the **Set Up Top-Level Design File** dialog box provide information about megafunctions created in or imported to the Pin Planner and allow you to make adjustments to connect megafunctions together. The **Direction** column indicates the direction of the port or port group as defined by the megafunction. The direction of a port cannot be changed.

The **Type** column indicates whether a port is available externally to the device. By default, all ports on all megafunctions created through the Pin Planner are of the **External** type, meaning they appear in the Groups List and All Pins List and can be assigned to I/O pins. You can change the port type by double-clicking the cell in the **Type column** for a port and selecting **Internal** or **External** from the list. Any ports on any megafunction connected to a port that has its type changed have their type changed to match automatically. This prevents internal and external megafunction ports from being connected to each other accidentally. Internal ports do not appear in the Groups List or All Pins List. If all the ports of a megafunction are internal, the megafunction does not appear in the Groups List.

The **Node Name** column is used to assign node names or device pins to ports. To connect a port to an existing location, double-click the cell in the **Node Name** column and select an existing node or device pin. To rename the selected port, enter a new node name in the **Node Name** column. This procedure only changes the name as it appears as a group member in the Groups List. To connect ports to each other, enter a node name that matches the node name of the ports of other megafunctions or an existing node.

Figure 5–10 shows an example of the port names of the megafunction named "output". When the port types and node names for both the input and output megafunctions are configured as in the two figures, they create a circuit similar to the one shown in Figure 5–11. In this way, you can connect megafunctions to each other and to other nodes in the design, improving the thoroughness of an I/O assignment analysis. This is especially useful for clock networks that are typically attached to multiple megafunctions or IP MegaCores.

Figure 5–10. Port Names of the Megafunction Named "Output"

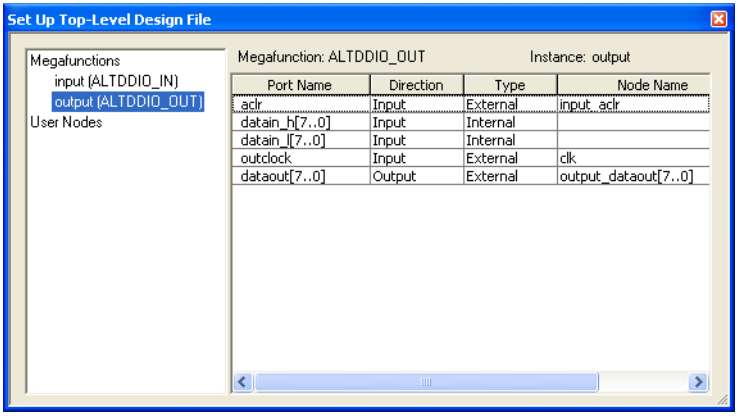
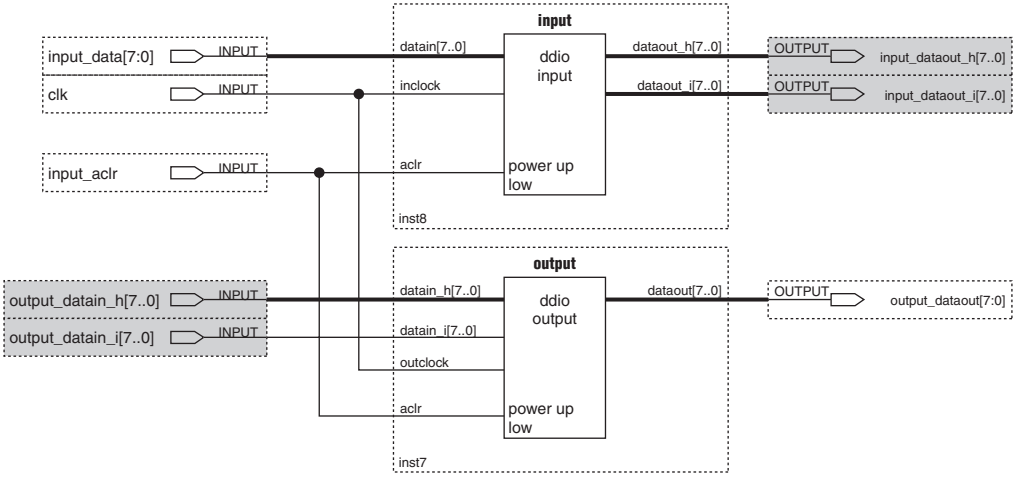


Figure 5–11. Schematic Representing Connections between Input and Output Megafunctions



Note to Figure 5–11:
(1) Gray pins indicate internal nodes of the design.

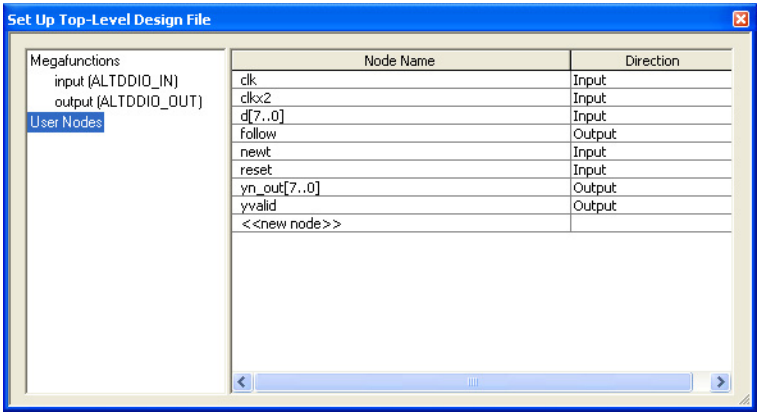
To edit the megafunction or IP MegaCore you created in the Pin Planner, select it in the Groups List. On the Edit menu, click **Edit Megafunction** to reopen the MegaWizard Plug-In Manager and make changes as necessary. If you make changes to a megafunction, you must import it again and reconfigure its node connections in the **Set Up Top-Level Design File** dialog box.

If you edit the megafunction outside of the Pin Planner, you must reimport its **.ppf** file to the Pin Planner.

Configure User Nodes for Creating a Top-Level Design File

Besides importing, creating, and configuring megafunctions for your project, you can also add your own nodes as top-level ports in your design. After you add user nodes, you can connect them to pins of the device. Creating and configuring user nodes early in the design cycle enables you to do early I/O analysis of these connections. Eventually, your user nodes will be connected to the internal logic of your design. In the **Set-Up Top Level Design File** dialog box, click **User Nodes** and enter new node information (Figure 5–12).

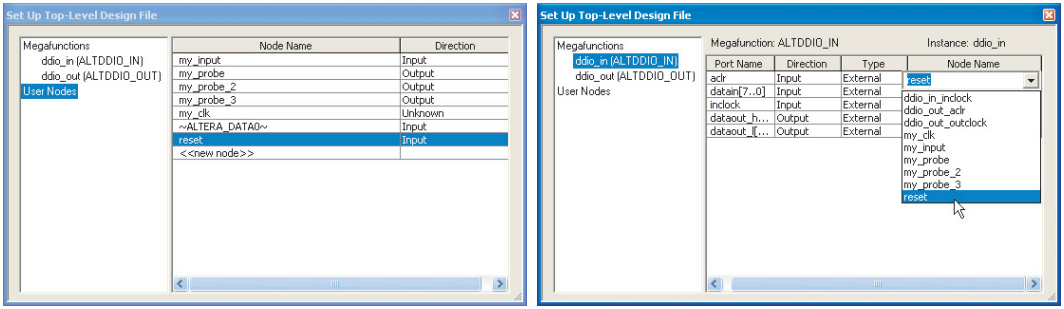
Figure 5–12. User Nodes



Each user node is associated with a node name and a direction. The direction is input, output, bidirectional, or unknown. If you do not select a direction, the node's direction is unknown. When you enter new node names in this window, the All Pins List and Groups List in the Pin Planner are also updated.

To create a new node, double-click in the **<<new node>>** row and enter the new node's name. If you have an existing top-level file, the ports of the top-level design file are always shown as existing user nodes (Figure 5–12). If you have no top-level design file, the newly created user nodes appear under the **User Nodes** list. The user nodes are also displayed when you configure node names for megafunctions. For example, to connect a user node named "reset" to a megafunction's reset input port, in the Node Name column, select "reset" to make this connection, as shown in Figure 5–13.

Figure 5–13. Connecting a User Node to a Megafunction Port



Create a Top-Level Design File

After you configure the user nodes, megafunctions, and IP MegaCores created in the Pin Planner, you can create a top-level design file in an HDL format. Use this file as the basis for the rest of your project, or use it to validate the I/O assignments already made.

To generate a top-level design file, right-click in the Package View and click **Create Top-Level Design File**. You can also generate a top-level file on the File menu by pointing to **Create/Update** and clicking **Create Top-Level Design File From Pin Planner**. The **Create Top-Level Design File** dialog box appears. Enter a name and select an HDL format (Verilog or VHDL). If the file already exists, you can choose to create a backup of the original file.

The top-level HDL file contains the external nodes of the megafunction and all other top-level user ports. The Pin Planner makes virtual pin assignments to internal nodes, so internal nodes are not assigned to device pins during compilation. If you use this top-level file as the basis for your project, internal megafunction ports must be connected to internal logic.



The top-level design file must be updated whenever changes are made to the design's top-level ports, including any node changes made in the **Set Up Top-Level Design File** dialog box.

Creating Pin-Related Assignments

A pin-related assignment is any assignment applied to a pin. For example, a pin-related assignment is a pin location assignment that assigns a design pin to a pin number (location) on the targeted device. Other pin-related assignments include assigning an I/O standard or current drive strength to a pin.

You can make pin-related assignments at any time during the design cycle, even before any design files have been developed. The accuracy and completeness of the pin-related assignments determines the accuracy of the I/O assignment analysis. If you do not have design files, create reserved pins to temporarily represent your top-level design I/O pins until the I/O pins are defined in your design files. If you do not have design files in your project, create an empty Verilog HDL or VHDL file with all the ports of the design defined.

Reserved pins are intended for future use but do not currently perform a function in your design. Reserved pins require a unique pin name and a pin location. Using reserved pins as place holders for future design pins increases the accuracy of the I/O assignment analysis.

The Quartus II software offers many tools and features for creating reserved pins and other pin-related assignments (Table 5–2). Each tool and feature is described in more detail in the following sections.

Table 5–2. Overview of Quartus II Tools and Features to Create Pin-Related Assignments (Part 1 of 2)

Feature	Overview
Pin Planner	• Make pin location assignments to one or more node names by dragging and dropping unassigned pins into the Package View • Edit pin location assignments for one or more node names by dragging and dropping groups of pins within the Package View • Visually analyze pin resources in the Package View • Display I/O banks and VREF groups • View the function of package pins using the pin legend • Make correct pin location decisions by referring to the Pad View window • Create, import, and edit megafunctions and IP MegaCores for early I/O planning • Generate a top-level wrapper file without design files based on early I/O assignments • Configure board trace models of selected pins for use in “board-aware” signal integrity reports generated with the **Enable Advanced I/O Timing** option
Assignment Editor	• Create and edit all types of pin-related assignments • Create and edit multiple assignments simultaneously with the **Edit** bar • Create pin assignments efficiently by viewing the different font styles used to display assigned and unassigned node names, as well as occupied and available pin locations • Provides user-selectable information about each pin, including the pad number, the t_{CO} requirement, and the t_{H} requirement

Table 5–2. Overview of Quartus II Tools and Features to Create Pin-Related Assignments (Part 2 of 2)

Feature	Overview
Tcl	• Create any pin-related assignments for multiple pins • Store and reapply all pin-related assignments with Tcl scripts • Make assignments from the command line
Chip Planner or Timing Closure Floorplan	• Create and change pin locations by dragging and dropping pins into the floorplan • Make correct pin location decisions by referring to the pad ID number and spacing • Display I/O banks, VREF groups, and differential pin pairing information
Synthesis Attributes	• Embed pin-related assignments using attributes in the design files to pass assignments to the Quartus II software

Using the Pin Planner

The Pin Planner is the main interface for creating and editing pin-related assignments. Use the Pin Planner Package View to make pin location and other assignments using a device package view instead of pin numbers. With the Pin Planner, you can identify I/O banks, VREF groups, and differential pin pairings to help you through the I/O planning process.

When planning your I/Os, it can be cumbersome to try to correlate pin numbers with their relative location on the package and their pin properties. The Pin Planner provides an intuitive graphical representation of the targeted device, also known as the Package View, that makes it easy to plan your I/Os, create reserved pins, and make pin location assignments. When deciding on a pin location, use the Pin Planner to gather information about available resources, as well as the functionality of each individual pin, I/O bank, and VREF group. You can assign locations to design pins by dragging and dropping each pin into the Package View.



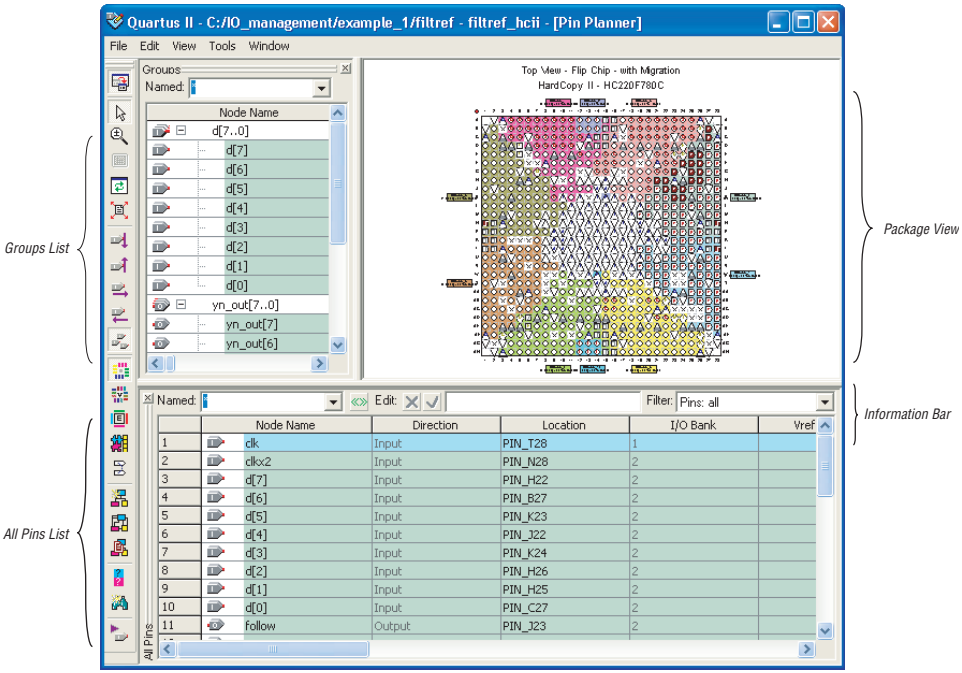
Maintaining good signal integrity (SI) requires that you follow pad distance and pin placement rules. Complementing the Pin Planner is the Pad View window, which displays the pads in order around the silicon die.

The Pin Planner includes the following sections (refer to Figures 5–14 through 5–19):

- "Groups List" on page 5–24
- "All Pins List" on page 5–27
- "Pad View Window" on page 5–30
- "Package View" on page 5–31
- "Pin Migration View" on page 5–34

- "Using the Pin Finder to Find Compatible Pin Locations" on page 5–36
- "Creating Reserved Pin Assignments" on page 5–38
- "Creating Pin Location Assignments" on page 5–39
- "Changing Pin Locations" on page 5–46
- "Show I/O Banks" on page 5–47
- "Show VREF Groups" on page 5–49
- "Show Edges" on page 5–51
- "Show DQ/DQS Pins" on page 5–53
- "Displaying and Accepting Fitter Placements" on page 5–54

Figure 5–14. Pin Planner



The Pin Planner feature supports cross-probing that allows you to select a pin in one view while simultaneously highlighting the pin in all of the different views. For example, if you select a pin in the Package View of the Pin Planner, the corresponding pad in the Pad View window is highlighted. If the pin has an assigned node name, the node name in the All Pins List and the Groups List is highlighted.

The Pin Planner and the Assignment Editor get their I/O Bank colors from Timing Closure Floorplan colors, which you can customize by performing the following steps:

1. On Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, under **Timing Closure Floorplan**, select **Colors**. The Colors page appears.
3. Make your color selections and click **OK**.

Groups List

The Groups List displays all of the buses from the top-level ports of your design and all the assignment groups in your project (Figure 5–15). Filter the group names displayed by typing in a wild card filter into the **Named** list. The Groups List allows you to create your own custom groups of pins and make location assignments to groups by dragging them into the Package View of the Pin Planner.

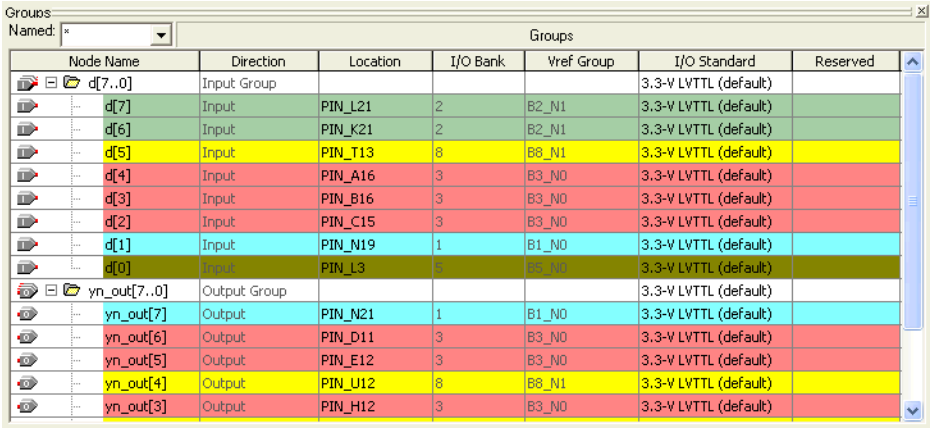


In the Groups List, all members of an assignment group are displayed, regardless of whether the member is a pin or an internal node.

The background color of pin locations in the Groups List easily identifies which pins belong to which I/O banks. The colors match the I/O bank colors in the Package View when **Show I/O Banks** is enabled. You can turn off the colors in both the Groups List and the All Pins List. On the Tools menu, click **Options**. In the **Category** list, select **Pin Planner**, and turn off **Show I/O bank color in lists**.

You can create and organize custom groups and group members in the **Assignment Groups** dialog box or directly in the Groups List in the Pin Planner. To open the **Assignment Groups** dialog box, on the Assignments menu, click **Assignment (Time) Groups**.

Figure 5–15. Groups List



To add a new group to the Groups List without opening the **Assignment Groups** dialog box, perform the following steps:

1. In the Groups List, in the **Node Name** column, double-click **<<new node>>**.
2. Type the group name.
3. Press **Enter**. The **Add Members** dialog box appears.
4. Type node names, wild cards, and assignment groups in the **Members** box, or browse to and select the node names from the **Node Finder** dialog box.
5. Click **OK**.

For more information about using Assignment Groups, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

You can also create a new group by selecting one or more node names within the Groups List or All Pins List. Right-click one of the selected node names, and on the right-click menu, click **Add to Group**.

As you plan your I/O placement, you may decide to add and remove members from a group.

To add a member to a custom group in the Groups List without opening the **Assignment Groups** dialog box, perform the following steps:

1. Right-click a group name in the Groups List and click **Add Members**.

2. Type in the name of the member or click the browse button to select one or more nodes from the **Node Finder** dialog box.

To remove a member from a group in the Groups List, perform the following steps:

1. Expand the group from which you want to remove a member.

2. Select one or more members that you want to remove.

3. Right-click the selected members, point to Edit and click **Delete**.

The Groups List provides many columns, some for information purposes and others to make assignments. You can edit the following columns only:

- Node Name
- Location
- I/O Standard
- Reserved
- Enable

Make changes to any of the values in these columns to adjust pin-related assignments. Other columns provide useful information during I/O planning, including the I/O Bank number, the VREF group, and the direction. To show or hide a column, right-click the column and click **Customize Columns**. You can also reorder and sort the columns from this menu.

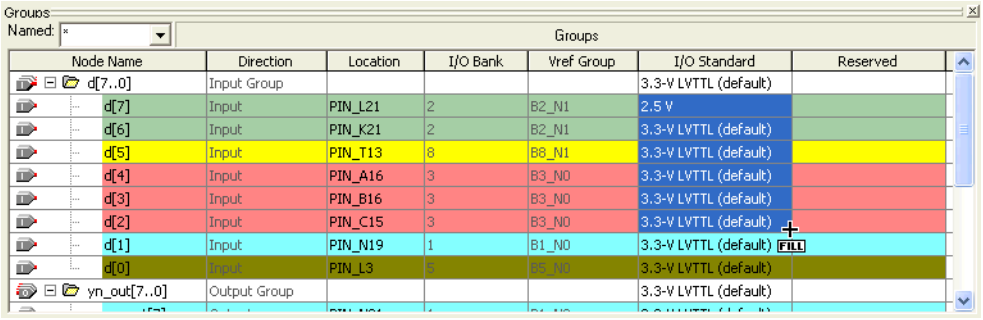


If an assignment group contains pins with different directions, the direction of the assignment group is a `bidir` group.

You can edit the columns in the Groups List in the same manner as a spreadsheet. You can copy and paste the Location, I/O Standard, and Reserved assignments to other rows in the list within the same column. You can also use Auto Fill to copy these assignments to other rows quickly.

To automatically fill a block of rows, set the desired assignment in one row and select the assignment's cell. Place the cursor over the lower right-hand corner of the cell until it changes to a cross with the word FILL (Figure 5–16). Click and drag up or down the column to select which cells to fill. When all the desired cells are selected, release the mouse button. The selected assignment is copied to all of the selected cells.

Figure 5–16. Auto Fill the Groups List

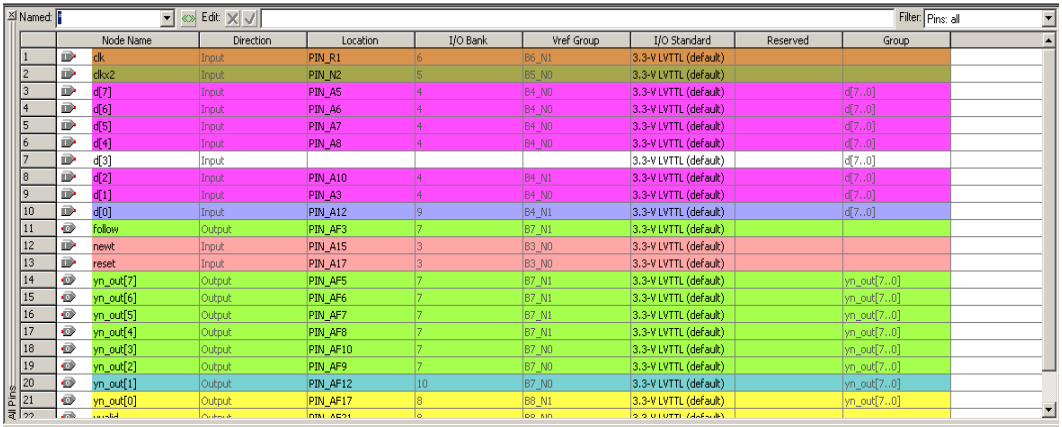


All Pins List

The All Pins List displays all of the pins in your design, including user-created pins (Figure 5–17). The All Pins list does not display buses; instead, it displays each individual pin of the bus. The background color of pin locations in the All Pins List easily identifies which pins belong to which I/O banks. The colors match the I/O bank colors in the Package View when **Show I/O Banks** is turned on. You can turn off the colors in both the All Pins List and the Groups List. On the Tools menu, click **Options**. In the **Category** list, select **Pin Planner**, and turn off **Show I/O bank color in lists**.

You must perform Analysis and Elaboration successfully to display pins in your design in the All Pins List. Individual user-reserved pins and nodes with pin-related assignments are always shown in the All Pins List.

Figure 5–17. All Pins List



	Node Name	Direction	Location	I/O Bank	Vref Group	I/O Standard	Reserved	Group
1	clk	Input	PIN_R1	6	B6_N1	3.3-V LVTTL (default)		
2	clkx2	Input	PIN_N2	5	B5_N0	3.3-V LVTTL (default)		
3	d[7]	Input	PIN_A5	4	B4_N0	3.3-V LVTTL (default)		d[7..0]
4	d[6]	Input	PIN_A6	4	B4_N0	3.3-V LVTTL (default)		d[7..0]
5	d[5]	Input	PIN_A7	4	B4_N0	3.3-V LVTTL (default)		d[7..0]
6	d[4]	Input	PIN_A8	4	B4_N0	3.3-V LVTTL (default)		d[7..0]
7	d[3]	Input				3.3-V LVTTL (default)		d[7..0]
8	d[2]	Input	PIN_A10	4	B4_N1	3.3-V LVTTL (default)		d[7..0]
9	d[1]	Input	PIN_A3	4	B4_N0	3.3-V LVTTL (default)		d[7..0]
10	d[0]	Input	PIN_A12	9	B4_N1	3.3-V LVTTL (default)		d[7..0]
11	follow	Output	PIN_AF3	7	B7_N1	3.3-V LVTTL (default)		
12	newt	Input	PIN_A15	3	B3_N0	3.3-V LVTTL (default)		
13	reset	Input	PIN_A17	3	B3_N0	3.3-V LVTTL (default)		
14	yn_out[7]	Output	PIN_AF5	7	B7_N1	3.3-V LVTTL (default)		yn_out[7..0]
15	yn_out[6]	Output	PIN_AF6	7	B7_N1	3.3-V LVTTL (default)		yn_out[7..0]
16	yn_out[5]	Output	PIN_AF7	7	B7_N1	3.3-V LVTTL (default)		yn_out[7..0]
17	yn_out[4]	Output	PIN_AF8	7	B7_N1	3.3-V LVTTL (default)		yn_out[7..0]
18	yn_out[3]	Output	PIN_AF10	7	B7_N0	3.3-V LVTTL (default)		yn_out[7..0]
19	yn_out[2]	Output	PIN_AF9	7	B7_N0	3.3-V LVTTL (default)		yn_out[7..0]
20	yn_out[1]	Output	PIN_AF12	10	B7_N0	3.3-V LVTTL (default)		yn_out[7..0]
21	yn_out[0]	Output	PIN_AF17	8	B8_N1	3.3-V LVTTL (default)		yn_out[7..0]

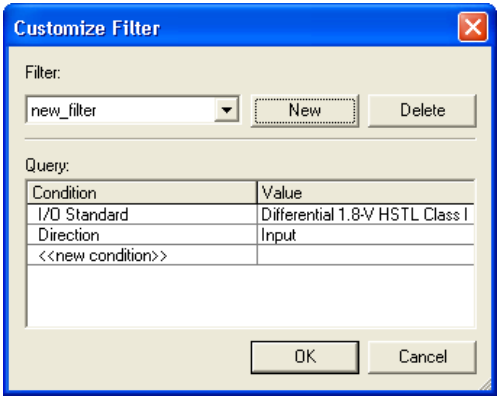
You can filter the list of pins in the All Pins List based on their node names by typing in a portion of the pin name in combination with wild card characters in the **Named** list. You can also filter the list of pins in the All Pins List based on the pins' attributes by selecting from the **Filter** list.

To create your own custom filter in the **Filter** list, specify a set of conditions from the following list:

- Assigned or unassigned
- Current strength
- Direction
- Edge location
- I/O Bank location
- I/O Standard
- VREF Group

To create a new filter in the All Pins List, in the **Filter** list in the All Pins List, select <<*new filter*>>. The **Customize Filter** dialog box appears (Figure 5–18).

Figure 5–18. Customize Filter Dialog Box



To create a custom filter for the All Pins List, perform the following steps:

1. In the **Customize Filter** dialog box, click **New**. The **New Filter** dialog box appears.
2. Enter the name of your custom filter in the **Filter name** text box.
3. You can base your new custom filter on existing filters by selecting from the **Based on Filter** list. If you do not want to base your custom filter on any other filter, select **Pins: all** from the **Based on Filter** list.
4. Click **OK**.
5. Add as many conditions as you require to the **Query** list. To add a condition, double-click **<<new condition>>** and select a condition from the **Condition** list. Select a value for the condition by double-clicking the cell next to your selected condition under the **Value** column.



To remove a condition from your filter, right-click the condition in the **Query** list and select **Delete**.

After specifying your conditions, the pins meeting the specified conditions are the only pins shown in the All Pins list. If the set of conditions contains a condition with more than one value, then the pins displayed must meet at least one of the values for that multiple-value condition.

To edit an existing custom filter, select **<<*new filter*>>** from the **Filter** list in the All Pins List. In the **Customize Filter** dialog box, select the custom filter you want to edit from the **Filter** list and add and remove conditions to the **Query** list.

Pins generated from a compilation or from a bus group are not editable. All other user-created pins are editable.

The All Pins List provides many columns, some for information purposes and others to make assignments. To show or hide a column, right-click the column heading and select **Customize Columns**. In addition, you can reorder and sort the columns from this menu.

You can edit the columns in the Groups List in the same manner as a spreadsheet. You can copy and paste assignments to other rows in the list within the same column. You can also use Auto Fill to quickly copy these assignments to other rows. To automatically fill a block of rows, set the desired assignment in one row and select the assignment's cell. Place the cursor over the lower right-hand corner of the cell until it changes to a cross with the word FILL as shown with the Groups List in Figure 5–16 on page 5–27. Click and drag up or down the column to select which cells to fill. When all the desired cells are selected, release the mouse button. The selected assignment is copied to all the selected cells.

Pad View Window

To maintain good signal integrity in designs, use the Pad View window to guide your pin placement decisions. Each device family is accompanied with pin placement rules, including pad spacing between various pin types.

For more information about pin placement rules, refer to the appropriate device handbook.

Edit or make pin assignments in the Pad View window by dragging and dropping a design pin into an available pad location.

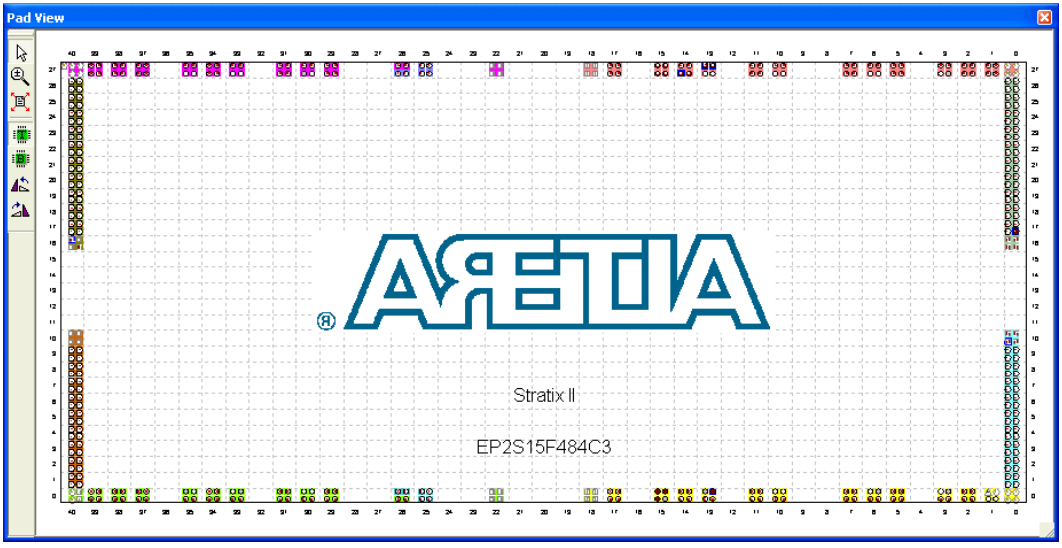
When you drag and drop a design pin into an available pad location, the corresponding pin number of the pad is assigned to the design pin. To assign a pad number to the design pin, perform the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. Click **Pin Planner** and turn on **Create pad assignment in the Pad View window**.

The column and row numbering around the Pad View window helps identify which pad row or pad column each pad is located. This is useful when the pin placement guidelines for your targeted device refer to pad rows and columns.

Since the Pad View window is a view of the I/O ring of the silicon within the package, flip chip packages appear inverted. Notice the reversed ALTERA logo in Figure 5–19. To understand the correlation between the package pins and the pads on the silicon die, the Pad View window and Package View are closely integrated together. When a pad is selected, the corresponding pin in the Package View is highlighted. Similarly, when a pin is selected in the Package View, the corresponding pad is highlighted in the Pad View window.

Figure 5–19. Pad View Window of a Stratix II Flip-Chip Device



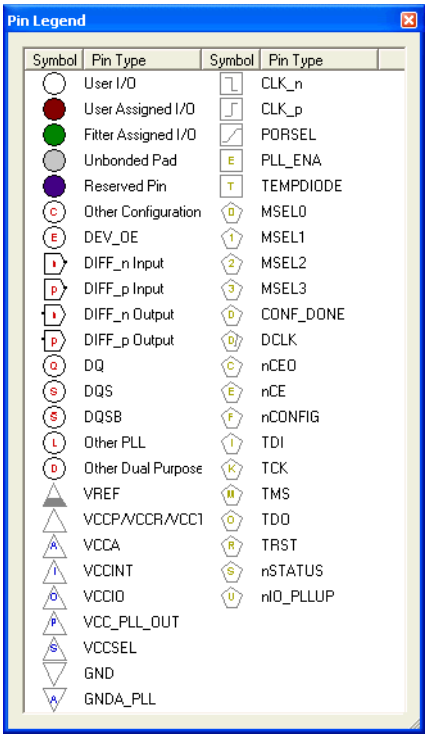
Package View

The Package View in the Pin Planner uses annotated pin symbols in different shapes and colors as visual representations of pins of the actual package (Figure 5–14 on page 5–23). The Package View eliminates the need to cross-reference each pin number with its physical location on the package described in the device package datasheet. When making pin location assignments in the Package View, switch between the different views to help you decide on a pin location. The different views in the Package View include I/O banks, VREF groups, Edges, DQ/DQS pins,

and differential pin pairs. For more information about the different views in the Package View, refer to the section in this chapter about the specific view you want to use. The sections are listed on page page 5–23.

The Pin Legend window provides a quick reference to the meanings of the pin symbol shapes, notations, and colors in the Package View. To view the Pin Legend window, on the View menu, click **Pin Legend Window** (Figure 5–20). You can also open the Pin Legend window from the Pin Planner toolbar or from the right-click menu in the Package View.

Figure 5–20. Pin Legend Window



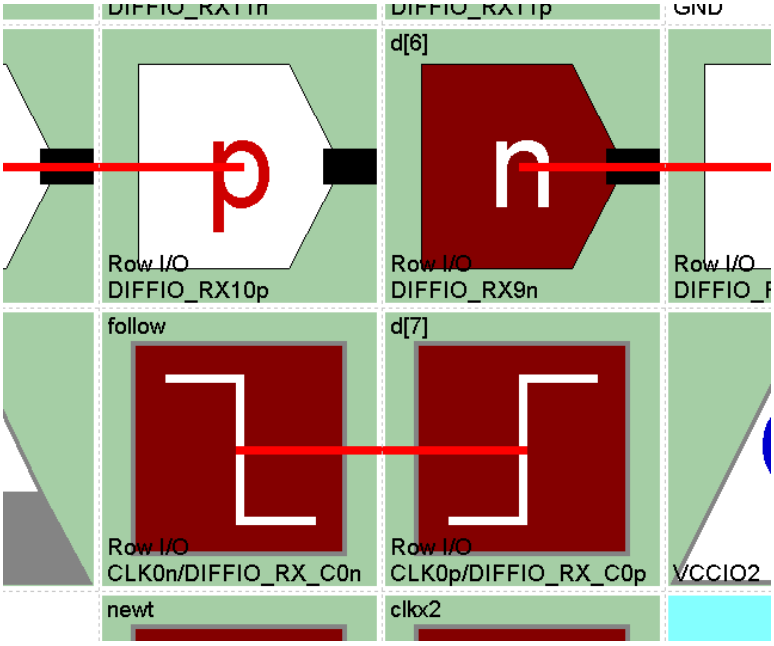
Planning your FPGA I/O assignments with your board design is necessary. If your FPGA device is oriented differently than in the Package View and Pad View window of the Pin Planner, rotate the Package View.

To rotate the Package View, on the View menu, point to Show and click **Rotate Left 90°** or **Rotate Right 90°** until your FPGA is shown in the desired orientation in the Package View. The red dot in the Package View

indicates the location of the first pin. For example, the red circle identifies where Pin A1 is located on a BGA package and where Pin 1 is located on a TQFP package.

You can also print the Package View with the pin names and pin types visible (Figure 5–21). To show the pin name (if available) or pin type for each pin in the Package View, on the View menu, click **Show Node Names** and **Show Pin Types**.

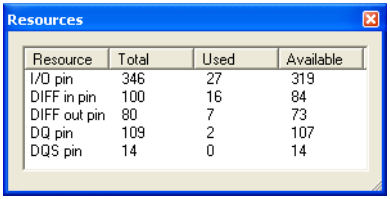
Figure 5–21. Package View with Show Node Names and Show Pin Types



To view pin resource usage, on the View menu, click **Resources Window**. The **Resources** dialog box appears (Figure 5–22).

For more detailed information about resources, view the Resource section of the Compilation Report.

Figure 5–22. Resources Window



If a HardCopy® II companion device is selected, the Pin Planner shows the Package View for the Stratix II device. To ensure correct pin migration between Stratix II and HardCopy II devices, run the I/O Assignment Analysis command or the Fitter.

Pin Migration View

The Pin Migration View in the Pin Planner shows the pins that change function in a migration device if you select one or more migration devices for your project. You can see changes for a pin by checking the **Show migration differences** box in the migration view. On the View menu, click **Pin Migration View** to open the Pin Migration View window. A pin is also highlighted in other views of the Pin Planner when you select any pin in the Pin Migration View.

The migration view provides detailed information about the pins which are affected in the migrated device. To select migration devices, perform the following steps:

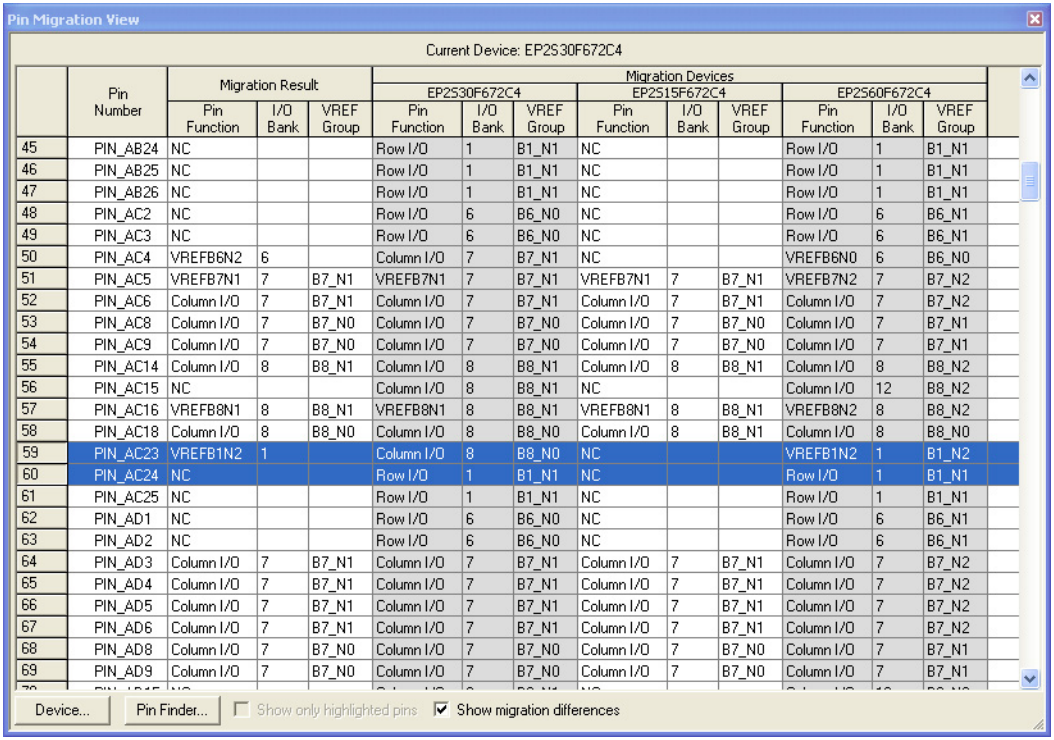
1. On the Assignments menu, click **Device**. The **Settings** dialog box appears.
2. Click **Migration Devices**. The **Migration Devices** dialog box appears.
3. Make your migration device selections and click **OK**.

The Pin Migration View helps you identify the difference in pins that can exist between migration devices. For example, in Figure 5–23, the highlighted pin `AC24` existed in the original EP2S30 device selected, but does not exist in one of the migration devices. Therefore, the migration result is a No Connect (NC). If you select your migration devices after you have successfully compiled a design and these migration devices have certain differences, an error occurs if you try to recompile your design.

For example, if you have a pin assignment in the original design, it might not be present in a migration device. You would have a successful fit if you had no migration devices selected. But if you select a migration device or devices for which the pin assignment cannot be honored because the pin does not exist in that device, an error occurs when you try to recompile. Therefore, Altera recommends you choose the supported migration devices early in the design planning process. When you select migration devices early in the design process, only the pins that exist in all migration devices are available in the Pin Planner and the Assignment Editor.

Additional differences may exist between migration devices, as shown in Figure 5–23.

Figure 5–23. Pin Migration View



	Pin Number	Migration Result			EP2S30F672C4			EP2S15F672C4			EP2S60F672C4		
		Pin Function	I/O Bank	VREF Group	Pin Function	I/O Bank	VREF Group	Pin Function	I/O Bank	VREF Group	Pin Function	I/O Bank	VREF Group
45	PIN_AB24	NC			Row I/O	1	B1_N1	NC			Row I/O	1	B1_N1
46	PIN_AB25	NC			Row I/O	1	B1_N1	NC			Row I/O	1	B1_N1
47	PIN_AB26	NC			Row I/O	1	B1_N1	NC			Row I/O	1	B1_N1
48	PIN_AC2	NC			Row I/O	6	B6_N0	NC			Row I/O	6	B6_N1
49	PIN_AC3	NC			Row I/O	6	B6_N0	NC			Row I/O	6	B6_N1
50	PIN_AC4	VREFB6N2	6		Column I/O	7	B7_N1	NC			VREFB6N0	6	B6_N0
51	PIN_AC5	VREFB7N1	7	B7_N1	VREFB7N1	7	B7_N1	VREFB7N1	7	B7_N1	VREFB7N2	7	B7_N2
52	PIN_AC6	Column I/O	7	B7_N1	Column I/O	7	B7_N1	Column I/O	7	B7_N1	Column I/O	7	B7_N2
53	PIN_AC8	Column I/O	7	B7_N0	Column I/O	7	B7_N0	Column I/O	7	B7_N0	Column I/O	7	B7_N1
54	PIN_AC9	Column I/O	7	B7_N0	Column I/O	7	B7_N0	Column I/O	7	B7_N0	Column I/O	7	B7_N1
55	PIN_AC14	Column I/O	8	B8_N1	Column I/O	8	B8_N1	Column I/O	8	B8_N1	Column I/O	8	B8_N2
56	PIN_AC15	NC			Column I/O	8	B8_N1	NC			Column I/O	12	B8_N2
57	PIN_AC16	VREFB8N1	8	B8_N1	VREFB8N1	8	B8_N1	VREFB8N1	8	B8_N1	VREFB8N2	8	B8_N2
58	PIN_AC18	Column I/O	8	B8_N0	Column I/O	8	B8_N0	Column I/O	8	B8_N1	Column I/O	8	B8_N0
59	PIN_AC23	VREFB1N2	1		Column I/O	8	B8_N0	NC			VREFB1N2	1	B1_N2
60	PIN_AC24	NC			Row I/O	1	B1_N1	NC			Row I/O	1	B1_N1
61	PIN_AC25	NC			Row I/O	1	B1_N1	NC			Row I/O	1	B1_N1
62	PIN_AD1	NC			Row I/O	6	B6_N0	NC			Row I/O	6	B6_N1
63	PIN_AD2	NC			Row I/O	6	B6_N0	NC			Row I/O	6	B6_N1
64	PIN_AD3	Column I/O	7	B7_N1	Column I/O	7	B7_N1	Column I/O	7	B7_N1	Column I/O	7	B7_N2
65	PIN_AD4	Column I/O	7	B7_N1	Column I/O	7	B7_N1	Column I/O	7	B7_N1	Column I/O	7	B7_N2
66	PIN_AD5	Column I/O	7	B7_N1	Column I/O	7	B7_N1	Column I/O	7	B7_N1	Column I/O	7	B7_N2
67	PIN_AD6	Column I/O	7	B7_N1	Column I/O	7	B7_N1	Column I/O	7	B7_N1	Column I/O	7	B7_N2
68	PIN_AD8	Column I/O	7	B7_N0	Column I/O	7	B7_N0	Column I/O	7	B7_N0	Column I/O	7	B7_N1
69	PIN_AD9	Column I/O	7	B7_N0	Column I/O	7	B7_N0	Column I/O	7	B7_N0	Column I/O	7	B7_N1

Notice that for PIN_AC23, the Migration Result for Pin Function is not an NC but a voltage reference VREFB1N2. This is because it is an NC in one of the migration devices, but a VREFB1N2 in the other migration device. In this type of result, VREF standards have a higher priority than an NC

and the result is VREFB1N2. You might not be making use of that pin for a port connection in your design, but you need to tie the VREF standard to supported standards on the actual board for the migration device.

If a migration device is selected, the Pin Planner shows only pins that are available for migration. Selecting a migration device allows you to either vertically migrate to a different density while using the same package, or migrate between packages with different densities and ball counts.

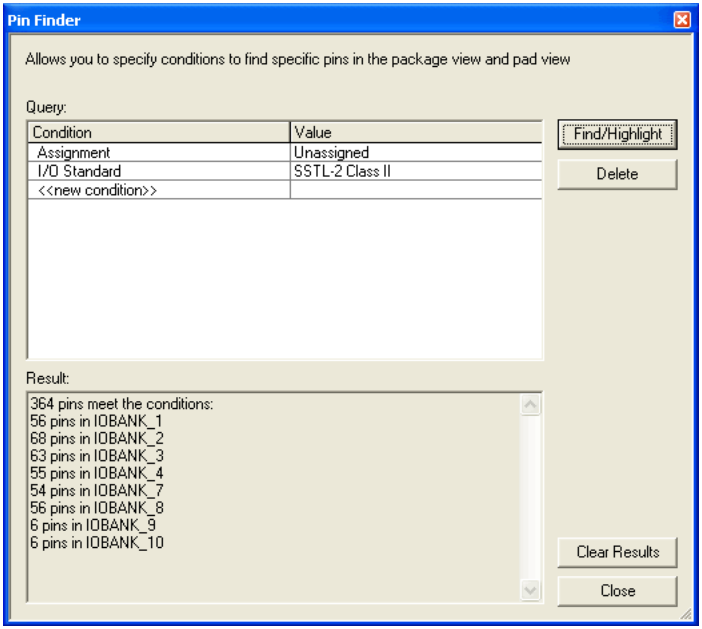
For more information about migration, refer to *AN90: SameFrame Pin-Out Design for FineLine BGA Packages*. For more information about designing for HardCopy II devices, refer to the *Quartus II Support for HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*.

Using the Pin Finder to Find Compatible Pin Locations

As FPGA pin-counts and I/O capabilities continue to increase, it becomes more difficult to understand the capabilities of each I/O pin and to correctly assign your design I/Os. To help you address this problem, the Pin Planner highlights all pins that match the list of conditions that you enter. To enter your conditions, perform the following steps with the Pin Planner open:

1. On the View menu, click **Pin Finder.** The Pin Finder window appears (Figure 5–24).

Figure 5–24. Pin Finder Window



2. In the Pin Finder window, create a list of conditions in the **Query** list.

   To add a condition to the **Query** list, double-click **<<new condition>>**, and select a condition from the list. Double-click the cell next to the new condition and select a desired value. For example, if you want to highlight all available pins that support the SSTL-2 Class II I/O standard, create an assignment condition and an I/O standard condition as shown in Figure 5–24.

   If the same condition type occurs more than once in the list, the Pin Finder searches for results that match any of its specified values. If you add more than one condition type, the Pin Finder searches for results that match all of the specified conditions.

3. In the Pin Finder window, click **Find/Highlight**. All of the pins that meet the specified conditions are highlighted in the Package View and in the Pad View window.

   At the same time, the **Results** list in the Pin Finder window displays a summary of the number of pins in each I/O Bank that meet the specified conditions.

Creating Reserved Pin Assignments

You can make reserved pin assignments to act as place holders for future design pins in the Package View or in the All Pins List. To create a reserved pin in the Package View, right-click an available pin, point to Reserve and click one of the available configurations.

When you reserve a pin from the Package View, the name of the reserved pin is set to `user_reserve_`<*number*> by default, and the pin symbol is filled with a dark purple color. The number increments by 1 for each additional reserved pin.

Alternately, you can reserve a pin in the All Pins List by performing the following steps:

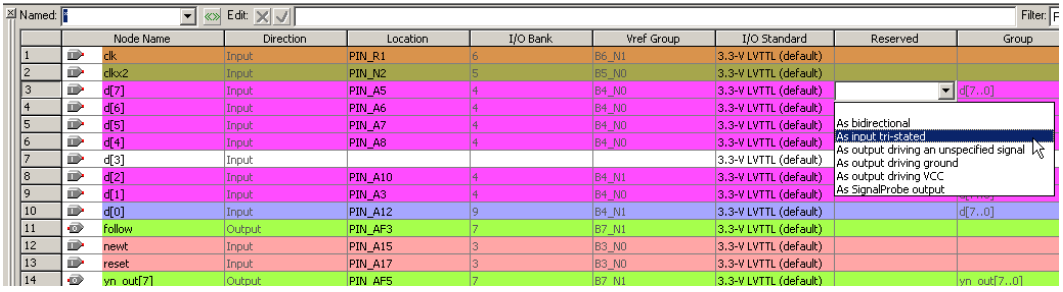
1. Type the pin name into an empty cell in the **Node Name** column. The pin name must not already exist in your design.

2. Select a pin configuration from the **Reserved** list (Figure 5–25).

   The following configurations are available:

   - Bidirectional
   - Input tri-stated
   - Output driving an unspecified signal
   - Output driving ground
   - Output driving VCC
   - SignalProbe output

Figure 5–25. Reserving a Pin in the All Pins List in the Pin Planner



Release reserved pins by selecting the blank entry from the **Reserved** list.

The **Direction** column is a read-only column and changes direction depending on the reserved selection.

Creating Pin Location Assignments

You can create pin locations assignments for one or more pins with the following methods:

- Assigning a location for unassigned pins
- Assigning a location for differential pins
- Assigning an unassigned pin to a pin location

You can disable or prevent any of these assignments using the **Enable** column in either the Groups List or the All Pins List. The **Enable** column is a special column that allows you to disable only the location assignment for a selected pin. Change the value of the cell in the **Enable** column for a selected pin from **Yes** to **No** by double-clicking the cell and selecting **No** from the list. A disabled pin only prevents location assignments when signals are assigned using drag and drop as described below. You can still make assignments directly in the **Location** columns in both the Groups List and All Pins List. To enable the location assignment again, change the **Enable** column back to **Yes**.

Assigning Locations for Unassigned Pins
To assign locations for all of your design pins, perform the following steps:

1. On the Edit menu, select an assignment direction.

   You can assign several pins simultaneously by choosing an assignment direction (Table 5–3). When assigning an entire bus, assignments are made in order from the most significant bit (MSB) to the least significant bit (LSB).

Table 5–3. Multiple Pins

Assignment	Pin Group
Assign Down	From the selected group of unassigned design pins, assign the pins downwards starting from the selected pin.
Assign Up	From the selected group of unassigned design pins, assign the pins upwards starting from the selected pin.
Assign Right	From the selected group of unassigned design pins, assign the pins from left to right starting from the selected pin.
Assign Left	From the selected group of unassigned design pins, assign the pins from right to left starting from the selected pin.
Assign One by One	Select a pin location for each of the design pins selected from the **Unassigned Pins** list.

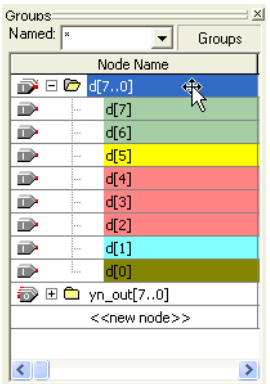
If there is an unassignable location in the path of the selected assignment direction, pins are assigned as far in the assignment direction as possible. Assign the rest of the pins in a separate location.

2. In the **Filter** list, select **Pins: unassigned**.

3. In the All Pins List, select one or more unassigned node names, or in the Groups List, select one or more buses.

   You can click on multiple node names using the control and shift keys. When you click on a pin or bus in the All Pins List or Groups List, the node name is highlighted and a crossing arrow displays above the cursor. Drag the selected cells into the Package View (Figure 5–26).

Figure 5–26. Drag Node Name in the Groups List



4. Drag and drop the selected pins or buses from the **All Pins** or Groups List to a location in the Package View.

Before you drag and drop your pins, you can optionally use the Pin Finder to locate pin locations that support your selected pins. When creating a query in the Pin Finder, add an Assignment condition and set it to **Unassigned**.

If you don't use the Pin Finder, you can drop pins directly into any of the following locations in the Pin Planner Package View: an available user I/O pin, I/O Bank, VREF Group, or Edge. On the View menu, you can display either I/O banks, VREF groups, or edges by going to the Show

submenu and toggling between **Show I/O Banks**, **Show VREF Groups**, and **Show Edges**. You can also toggle between these views from the Pin Planner toolbar or from the right-click menu in the Package View.

Available single-ended user I/O pins are represented by empty circles in the Package View. The letter inside the circle provides information about the user I/O pin. Negative and positive differential pins are shaped like hexagons and contain the letters "n" and "p", respectively. For a complete listing of I/O pin shapes, notations, and colors, from the View menu, toolbar, or Package View right-click menu, open the Pin Legend window.

In the Package View, I/O banks are displayed as rectangles labeled `IOBANK_<`*number*`>` (Figure 5–31 on page 5–48). In each I/O bank, there are one or more VREF groups. VREF groups are displayed as rectangles labeled `VREFGROUP_B<`*I/O Bank number*`>_N<`*index*`>` (Figure 5–33 on page 5–50).

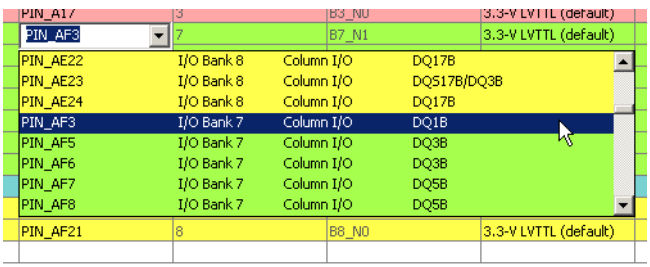
Edge locations are displayed as rectangles labeled `EDGE_<`*direction*`>`. To make an edge assignment, drag and drop pins into one of the four edges, `EDGE_TOP`, `EDGE_BOTTOM`, `EDGE_LEFT`, or `EDGE_RIGHT`.



You can drag and drop pins from the **Node Finder** dialog box or from the Block Diagram/Schematic File into the Package View.

Click on the **New Node** button () in the All Pins List to jump directly to the new node row without scrolling all the way down. When you click on the **Location Assignment** cell in the All Pins List, a drop-down combo box with all the assignable pins is opened (Figure 5–27).

Figure 5–27. Combo Box in the Pin Planner



The combo box shown in Figure 5–27 displays each row in a color matching its I/O bank color. In the combo box, each pin location row displays the location assignment column, its I/O bank column, and its special function column. While making assignments for a node, if you

double click the **Edit** field in the All Pins List, a pull-down box appears with all the possible values for the assignment.Choose a value and click the green check button ( ) to make the assignment or click the red cross button ( ) to cancel the assignments change.

Assigning a Location for Differential Pins

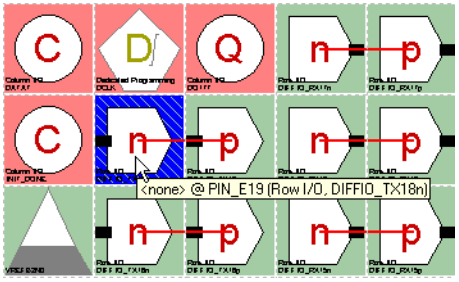
To identify and assign differential pins using the Pin Planner, perform the following steps:

1. On the View menu, click **Show Differential Pin Pair Connections**.

   A red line connects the positive and negative pins of the differential pin pairing. The positive and negative pins are labeled in the Package View with the letters "p" and "n", respectively (Figure 5–28).

2. Use the tool tips to identify LVDS-compatible pin locations by holding the mouse pointer over a differential pin in the Package View (Figure 5–28).

Figure 5–28. Tool Tip of a Differential Pin



The tool tip shows the design pin name and pin number, as well as its general and special functions.

The tool tip for differential receiver and transmitter channel pins that are also available as user I/O is shown in the following format:

<design pin name> `@ PIN_`*<Package Pin Number>* `(`*<Row|Column>* `I/O, DIFFIO_`*<RX/TX>*`<`*differential pin pair number*`><p|n>)`

The tool tip for dual-purpose LVDS I/O channel pins is shown in the following format:

<design pin name> `@ PIN_`*<Package Pin Number>* `(`*<Row|Column>* `I/O, LVDS`*<differential pin pair number>*`<p|n>)`

3. From the All Pins List or Groups List, click on the differential pin.

4. From the All Pins List or Groups List, drag and drop the selected pin to a differential positive pin location in the Package View.

   Optionally, before you drag and drop your pins, you can use the Pin Finder to locate pin locations that support your selected pins. When creating a query in the Pin Finder, add an assignment condition set to **Unassigned** and an I/O standard condition set to your differential I/O standard.

   The unassigned differential pin that you drag to the Package View represents the positive pin of the differential pair. The Fitter recognizes the negative pin of the differential pair automatically and creates it in the PIN file.

   If you assign a differential pin to a pin location, the negative pin becomes unassignable. The Quartus II software recognizes the negative pin as part of the differential pin pair assignment. However, the assignment is not entered in the QSF.

   If you have a single-ended clock that feeds a PLL, assign the pin only to the positive clock pin in the targeted device. Single-ended pins that feed a PLL and are assigned to the negative clock pin in the targeted device cause the design to fail to fit.

For more information about the general and special functions displayed by the tool tip, refer to the Device Pin-Outs available at **www.altera.com**.

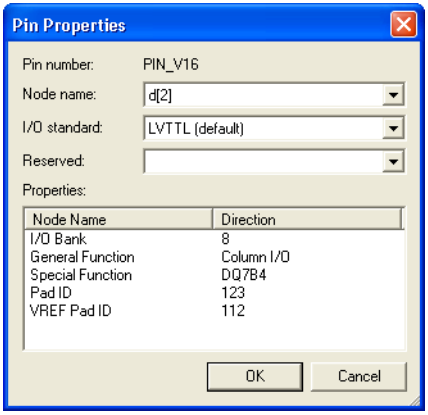
Assigning an Unassigned Pin to a Pin Location

Use the following steps to select a pin location and assign a design pin to that location:

1. In the Package View, select an available pin location.

2. On the View menu, click **Pin Properties.** The **Pin Properties** dialog box appears (Figure 5–29).

Figure 5–29. Pin Properties Dialog Box



You can use the **Pin Properties** dialog box to create pin location and I/O standard assignments. The **Pin Properties** dialog box also displays the properties of the pin location, including the pad ID (Table 5–4). The pad ID is important information when following pin spacing guidelines. Adjacent pin numbers do not always represent adjacent pads on the die. Use the Pad View window to help correlate pad location and the distance between your user I/O pins and VREF pins.

3. Select a pin from the **Node Name** list.
4. To assign or change the I/O standard, select an I/O standard from the **I/O standard** list.
5. Click **OK**.

For more information about pin placement, refer to the appropriate device handbook.

Table 5–4 provides a description of each field in the **Pin Properties** dialog box.

Table 5–4. Pin Properties

Pin Property	Description
Pin Number	Pin number used in the package *(1)*
Node Name	Node name assigned to the pin location
I/O Standard	I/O standard assigned to the pin name and location
Reserved	If reserved, determines how to reserve this pin
I/O Bank	I/O bank number of the pin
General Function	General function of the pin (row/column I/O, dedicated clock pin VCC, GND)
Special Function	Special function of the pin (LVDS, PLL)
Pad ID	Pad number connected to pin
VREF Pad ID	The pad ID for the V_{REF} pin used for voltage referenced I/O standards

Note to Table 5–4:
(1) For more information about how pin numbers are derived, refer to the device pin-out on the Altera website, www.altera.com.



You can also open the **Pin Properties** dialog box by double-clicking on a pin in the Package View of the Pin Planner, or by right-clicking the pin in the Package View of the Pin Planner, and clicking **Pin Properties**.

Error Checking Capability

The Pin Planner has basic pin placement checking capability, preventing pin placements that violate the fitting rules. The following checks are performed by the Pin Planner as you make pin-related assignments:

- An I/O bank or VREF group is an unassignable location if there are no available pins in the I/O bank or VREF group.
- The negative pin of a differential pair is unassignable if the positive pin of the differential pair has been assigned with a node name with a differential I/O standard.
- Dedicated input pins (for example, dedicated clock pins) are an unassignable location if you attempt to assign an output or bidirectional node name.
- Pin locations that do not support the I/O standard assigned to the selected node name become unassignable.
- All nodes in the same VREF group must have the same VREF voltage. Apply this only to HSTL- and SSTL-type I/O standards.



To perform a more comprehensive check on your pin placements, perform I/O assignment analysis.

For more information about assignment analysis, refer to "Using I/O Assignment Analysis to Validate Pin Assignments" on page 5–67.

To display live information about the warnings and errors in your pin-related assignments, enable the live I/O check feature in the Quartus II software. For more information about the live I/O check feature, refer to the section "Using the Live I/O Check Feature to Validate Pin Assignments" on page 5–65.

After creating a pin location, the **Location**, **I/O Bank**, and **VREF Group** columns are populated in both the All Pins List and the Groups List. In the Package View, the occupied pins are filled with a dark brown color.

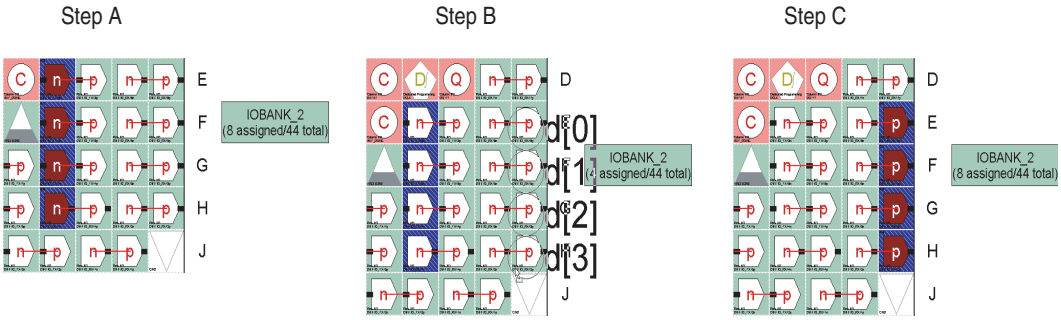
Changing Pin Locations

The Pin Planner allows you to change the location of multiple pins simultaneously. To change pin locations, select one or more pins in the Package View or Pad View window, and drag the pins to a new location.

You can change pin locations more quickly and easily if you understand which user I/O pins are available and where they are, physically, on the device. For example, in the Package View, you can move a column of pins closer to the edge of the device for easier PCB routing (Figure 5–30). In this example, you are moving multiple I/O pins to the area closest to the edge of the I/O bank. To change pin locations, perform the following steps:

1. In the Package View, select multiple pins by holding down the left mouse button and dragging over the pins you want to move (Figure 5–30, step A).
2. Drag the group of pins to the area of placement (Figure 5–30, step B).
3. Drop the pins into the area closest to the edge of the I/O bank (Figure 5–30, step C).

Figure 5–30. Changing the Locations for a Group of Pins

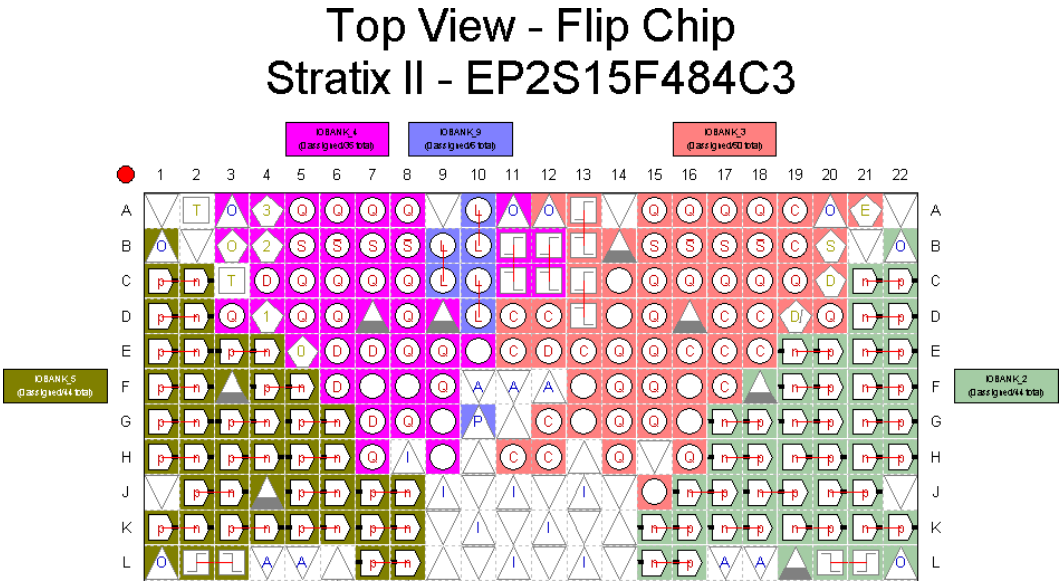


Show I/O Banks

When you turn on **Show I/O Banks** in the View menu, in the Show submenu, or on the right-click menu in the Package View, the Package View groups I/O pins that share the same `VCCIO` pin using different colors (Figure 5–31). When planning your I/O pins, it is important to guide your pin placement decisions by placing pins with compatible I/O standards into the same I/O bank. For example, you cannot place an `LVTTL` pin with an I/O standard of LVTTL in the same bank as another pin with an I/O standard of 1.5 V HSTL Class I.

For more information about compatible I/O standards, refer to the appropriate device handbook.

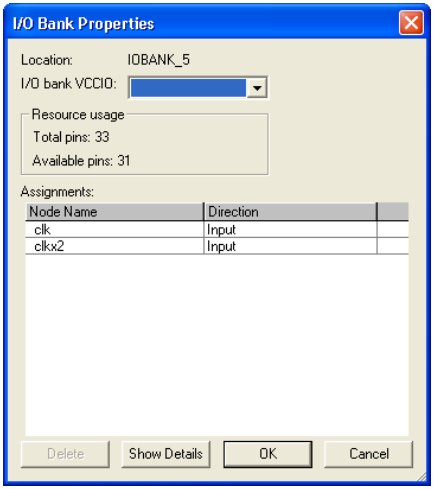
Figure 5–31. Package View with I/O Banks



When you turn on **Show I/O Banks**, the Package View allows you to view the properties of each I/O bank. Select an I/O bank in the Package View. On the View menu, click **I/O Bank Properties.** The **I/O Bank Properties** dialog box appears (Figure 5–32). The **I/O Bank Properties** dialog box lists all node names assigned to that I/O bank.

To view all node names that are assigned within the I/O bank, click **Show Details** in the **I/O Bank Properties** dialog box. You can also assign the V_{CCIO} for the I/O bank by selecting a voltage from the **I/O bank VCCIO** list.

Figure 5–32. I/O Bank Properties



Under **Resource Usage**, the total number of pins in the I/O banks is displayed, including assignable and unassignable pins, as well as the total number of available assignable pins. Adjust the intensity of colors of the I/O banks in the Package View by performing the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, select **Pin Planner**. The **Pin Planner** page appears.
3. Under **I/O banks color setting**, adjust the **I/O bank color intensity** using the slide bar.
4. Click **OK**.

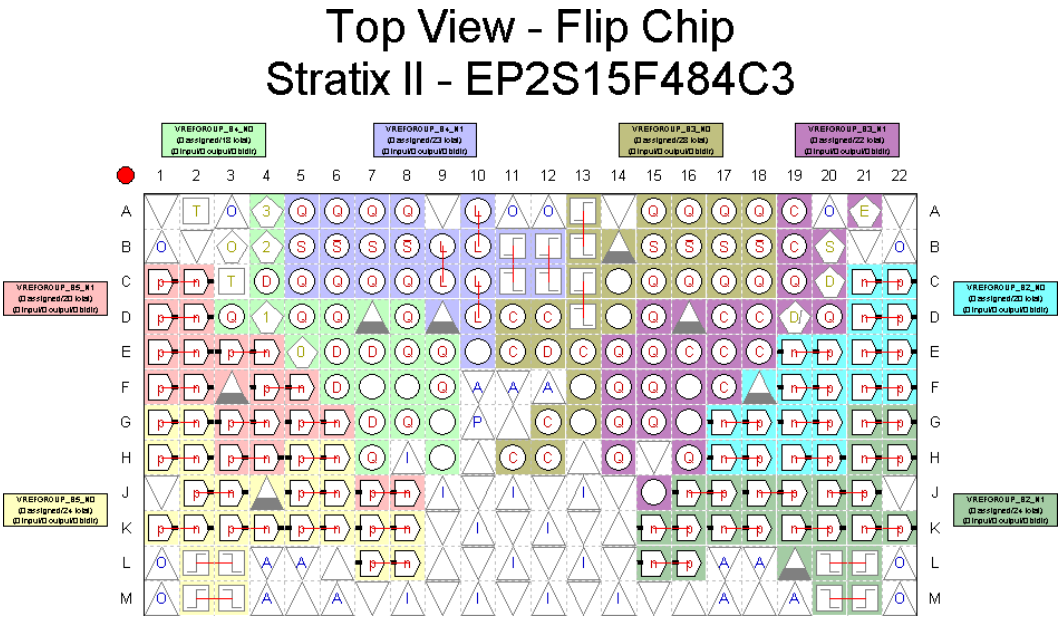
Show VREF Groups

You can use different colors to indicate different groups of I/O pins sharing the same VCCIO and VREF pins in the package view (Figure 5–33). When planning your I/O pins, it is important to place pins with compatible voltage-referenced I/O standards in the same I/O bank. To guide your pin placement decisions by placing compatible I/O standards requiring VREF pins into the same VREF group, on the View menu, point to Show and click **Show VREF Groups**. For example, pins with I/O standards SSTL-18 Class II and 1.8V-HSTL Class II are

compatible and can be placed into the same VREF group. It is also important to be aware of the number and direction of pins within a VREF group for simultaneous switching noise (SSN) analysis.

For more information about compatible I/O standards, refer to the appropriate device handbook.

Figure 5–33. Package View with VREF Groups

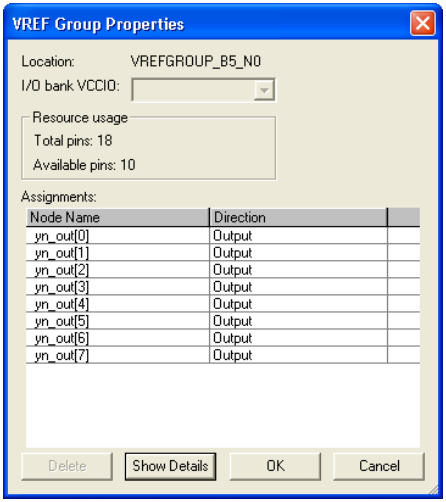


When you turn on **Show VREF Groups**, the Package View allows you to show the properties of each VREF group. Select a VREF group in the Package View, and on the View menu, click **VREF Group Properties.** The **VREF Group Properties** dialog box appears (Figure 5–34).

In the **VREF Group Properties** dialog box, all node names assigned to the VREF group are listed. Click **Show Details** to view node names that are assigned to pin numbers within the VREF group.

Any design pins that are assigned to the VREF group and not to a pin number are listed in the **Assignments** list. The Resource usage section describes the total number of pins in the VREF group and the total number of available assignable pins. It also keeps a running tally of the input, output, and bidirectional pins.

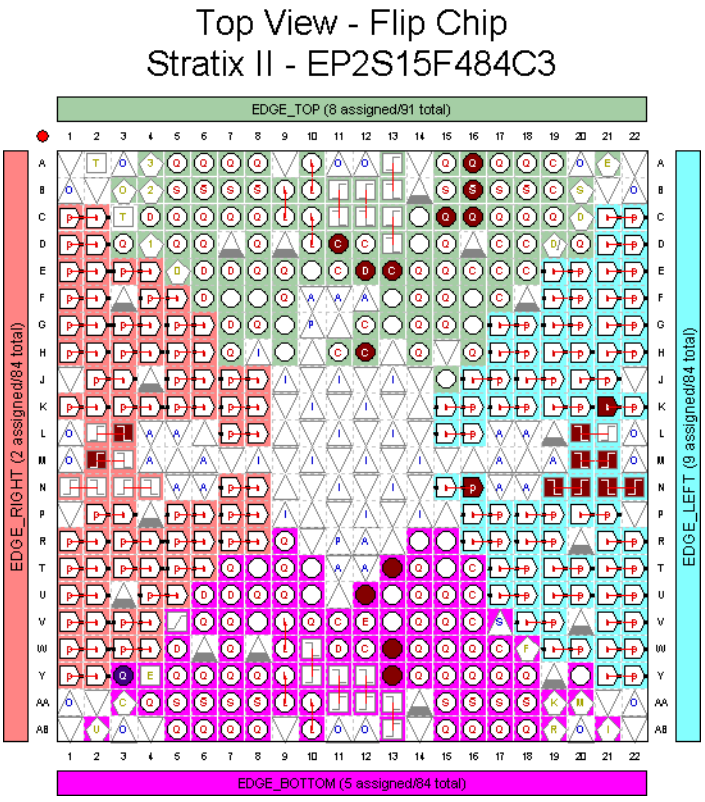
Figure 5–34. VREF Group Properties



Show Edges

You can use different colors to indicate the four edges of the package in the Package View (Figure 5–35). To do this, on the View menu, point to show and click **Show Edges**, or from the right-click menu, click **Show Edges**. If the exact location of a pin is not a priority when planning your I/O pins, use an Edge assignment.

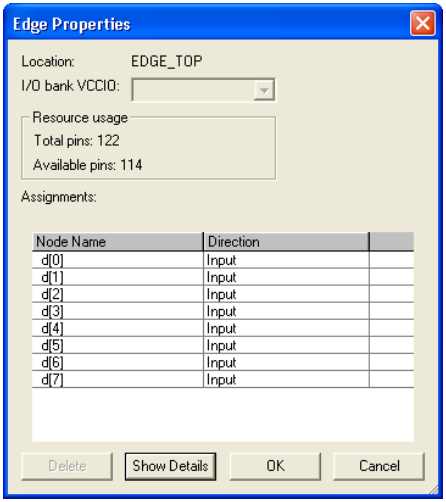
Figure 5–35. Package View with Edges



When you turn on **Show Edges**, the Package View allows you to show the properties of each Edge. Select an Edge in the Package View. On the View menu, click **Edge Properties.** The **Edge Properties** dialog box appears.

In the **Edge Properties** dialog box, all node names assigned to the Edge are listed (Figure 5–36). To view all node names assigned to a pin number within an Edge, in the **Edge Properties** dialog box, click **Show Details**.

Figure 5–36. Edge Properties

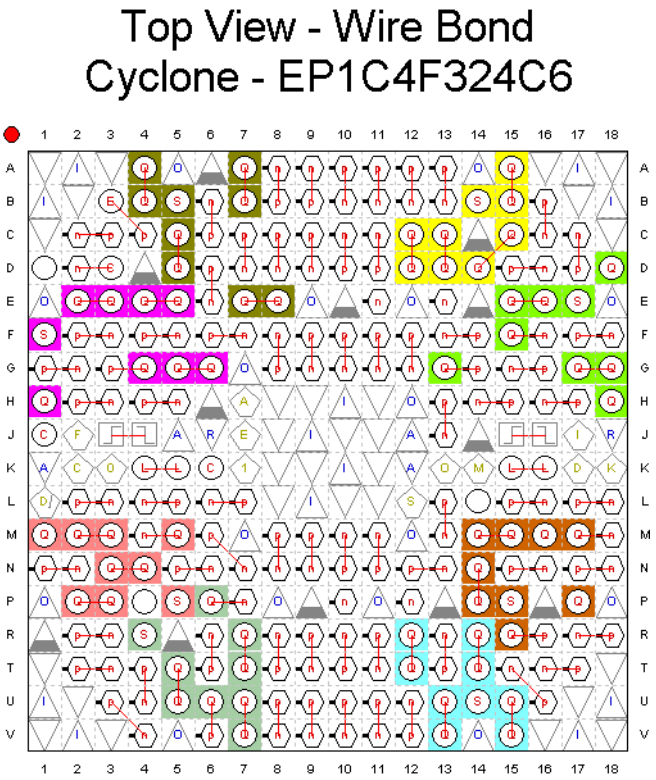


Show DQ/DQS Pins

You can use different colors to highlight groups of DQ and DQS pins in the Package View(Figure 5–37). To do this, on the View menu, point to Show and click **Show DQ/DQS Pins**, or from the right-click menu, click **Show DQ/DQS Pins**. Highlighting these DQ/DQS groups easily identifies which DQ pins are associated with a specific DQS strobe pin. Select between the following DQ/DQS modes:

- ×4 Mode
- ×8/×9 Mode
- ×16/×18 Mode
- ×32/×36 Mode

Figure 5–37. DQ/DQS Pins (1)



Note to Figure 5–37:
(1) This DQ/DQS view shows an ×8 mode.

For example, when implementing DDR II in a Stratix II device, there are dedicated pins designed specifically to be used as DQ and DQS pins.

For information about using the altdq and altdqs megafunctions to configure DQ and DQS pins, refer to the *altdq & altdqs Megafunction User Guide*.

Displaying and Accepting Fitter Placements

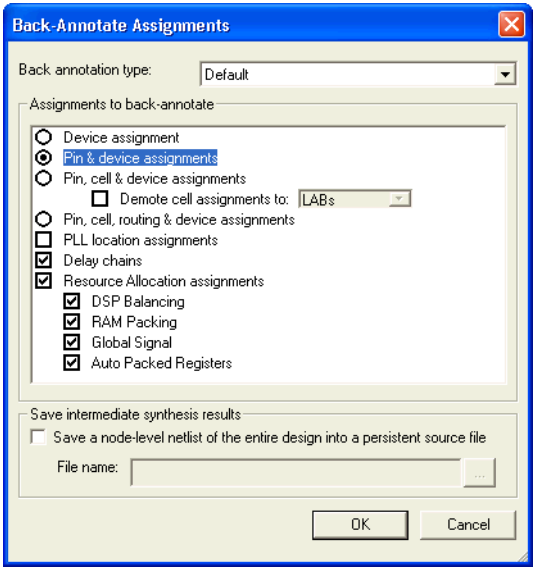
In addition to the **Show I/O Banks**, the **Show VREF Groups**, and the **Show Edge** views, you can also show pins placed by the Fitter. To display these pins, on the View menu, or in the Pin Planner toolbar, or on the right-click menu in the Package View, point to Show and click **Show Fitter Placements**.

The Fitter provides optimal placement to unassigned pins based on design constraints when you perform a compilation or an I/O Assignment Analysis. When you choose **Show Fitter Placements**, the Fitter-placed pins are shown as green-filled pins in the Package View. You can create a copy of the Fitter placements in your project QSF using the Back-Annote Assignments command.

To create assignments for all Fitter-placed pins into your project QSF, perform the following steps:

1. On the Processing menu, click **Start Compilation**, or on the Processing menu, point to Start and click **I/O Assignment Analysis**.
2. On the Assignments menu, click **Pin Planner**. The Pin Planner appears.
3. On the View menu, point to Show and click **Show Fitter Placements**. You can also access this command from the Pin Planner toolbar or on the right-click menu in the Package View.
4. Review the Fitter placements.
5. To create location assignments for these Fitter placements, perform the following steps:
    a. On the Assignments menu, click **Back-Annotate Assignments**. The **Back-Annotate Assignments** dialog box appears.
    b. Select **Pin & device assignments** (Figure 5–38).
    c. Click **OK**.

Figure 5–38. Back-Annotate Assignments Dialog Box



To create assignments for a selection of the Fitter-placed pins, perform the following steps:

1. On the Processing menu, click **Start Compilation**, or on the Processing menu, point to Start**,** and click **I/O Assignment Analysis**.
2. On the Assignments menu, click **Pin Planner**.
3. On the View menu, point to Show and click **Show Fitter Placements**, and review the placements.
4. In the Pin Planner, select one or more Fitter-placed pins for which you want to create assignments.
5. Right-click one of the selected pins, and click **Back Annotate**.
6. On the File menu, click **Save Project**. The Assignments are written to the QSF.

For more information about how the Quartus II software writes and updates the QSF, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

Altera recommends you use the Pin Planner to create and edit pin-related assignments. However, you may find some of the other tools provided for use with the Quartus II software to be useful for working with pin-related assignments. The following sections describe these tools.

Assignment Editor

The Assignment Editor provides a spreadsheet-like interface that allows you to create and change all pin-related assignments.

Two methods are available for making pin assignments with the Assignment Editor. The first involves selecting from all assignable pin numbers of the device and assigning a pin name from your design to this location.

The second involves selecting from all pin names in your design and assigning a device pin number to the design pin name. In either method, take advantage of row background coloring (pin numbers within the same I/O bank have a common background color), auto fill node names, and pin numbers to assist in making your assignments.

Setting Pin Locations from the Device Pin Number List

It is important to understand the properties of a pin location before assigning that location to a pin in your design. For example, you must know which I/O bank or VREF group the pin belongs to when following pin placement guidelines.



For more information about pin placement guidelines, refer to the appropriate device handbook.

Before creating pin-related assignments, perform analysis and elaboration or analysis and synthesis on your design to create a database of your design pin names. Then perform the following steps:

1. To open the Assignment Editor, on the Assignments menu, click **Assignment Editor.**

2. In the **Category list**, select **Pin**.

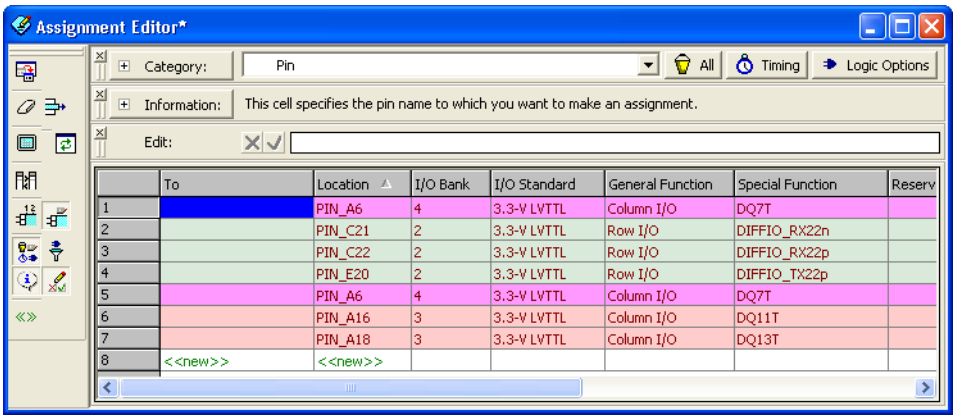
   Creating pin assignments can be difficult when you need to check which I/O bank the pin belongs to or which VREF pad the pin uses. By selecting the **Pin** category, more pin-related information is visible in the spreadsheet to help you create pin location assignments.

The Assignment Editor does not show assignments to individual nodes made with wildcards or assignment groups.

3. On the View menu, click **Show All Assignable Pin Numbers**.

You can also view all assignable pins in the All Pins List in the Pin Planner. Right-click anywhere in the Groups List or All Pins List, and click **Show Assignable Pins**. When the All Pins List filter is set to **Pins: unassigned** or **Pins: all**, a list of all assignable pin numbers for the targeted device is shown in the **Location** column (Figure 5–39).

Figure 5–39. Assignment Editor with Show All Assignable Pin Numbers



4. Find a pin number in the spreadsheet. In the same row, double-click the cell in the **To** column. Type the pin name or select a pin from the pull-down list. If analysis and elaboration has been performed, your design pins are listed in the pull-down list.

As you type in a pin name, the Assignment Editor automatically completes the field by looking up the pin names stored in the database created from the initial analysis and elaboration. Pin names already assigned to a pin location are shown in italics.

Setting Pin Locations from the Design Signal Name List

It is important to understand the properties of a pin location before assigning that location to a pin in your design. For example, you must know which I/O bank or VREF group the pin belongs to when following pin placement guidelines.

For more information about pin placement guidelines, refer to the appropriate device handbook.

To set the pin locations from the design pin name list, perform the following steps:

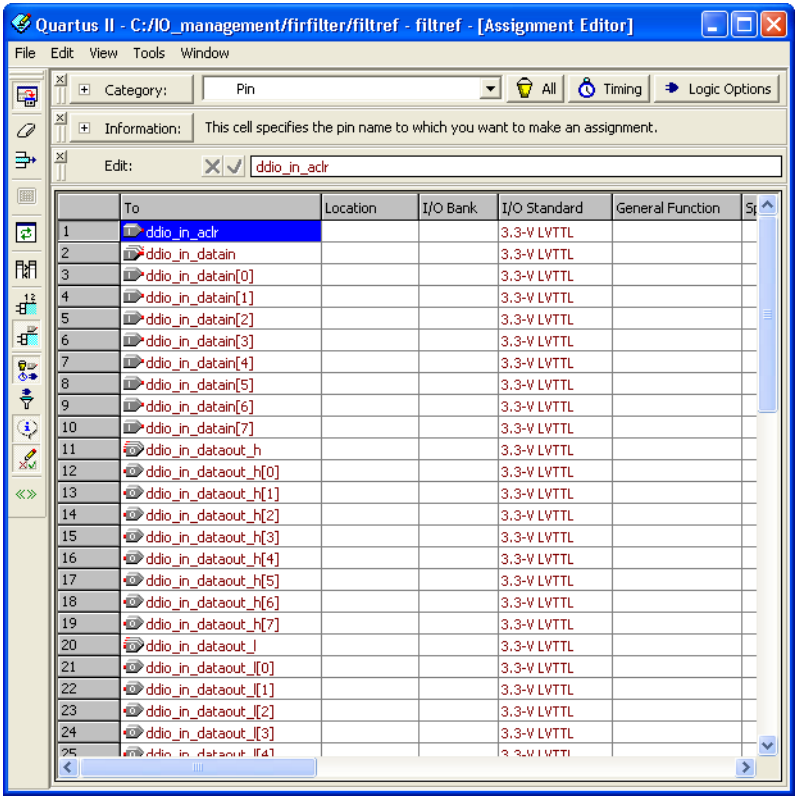
1. To open the Assignment Editor, on the Assignments menu, click **Assignment Editor**.
2. In the **Category** list, select **Pin**.

   Creating pin assignments can be difficult when you have to check which I/O bank the pin belongs to, or which VREF pad the pin uses. By selecting the **Pin** category, more pin-related information is visible in the spreadsheet to help you create pin location assignments.

   The Assignment Editor does not show assignments to nodes made with wildcards or assignment groups.
3. On the View menu, click **Show All Known Pin Names**.

   A list of all pin names in your design is shown in the **To** column (Figure 5–40).

Figure 5–40. Assignment Editor with Show All Known Pin Names



To list a selection of pin names from your design into the spreadsheet of the Assignment Editor, type the pin names with or without wild cards into the **Node Filter** bar. This is effective when you want to assign common pin-related assignments to a selection of pins in your design.

For more information about using the **Node Filter** bar, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

4. Find a pin name in the spreadsheet, and double-click the cell in the same row of the **Location** column. Select a pin number from the pull-down list which contains all assignable pin numbers in the selected device. You can also start typing the pin number and let the **Assignment Editor** automatically complete it for you. Instead of typing `PIN_AA3`, type `AA3` and let the Assignment Editor auto complete the pin number to `PIN_AA3`.



Pin locations that already have a pin name assignment appear in the Assignment Editor in italics.



For more information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

Tcl Scripts

Tcl scripting allows you to write scripts to create pin-related assignments. To run a Tcl script with your project, type the following command at a system prompt:

```
quartus_sh -t my_tcl_script.tcl ↵
```

You can also type individual Tcl commands into the Tcl console window. To use the Tcl console, on the View menu, point to Utility Windows and click **Tcl Console**. In the Tcl Console window, type your Tcl commands. The following example shows a list of Tcl commands that creates pin-related assignments to the input pin `address[10]`.

Example 5–1. Tcl Commands to Create Pin-Related Assignments

```
set_location_assignment PIN M20 -to address[10] -comment"Address pin to Second FPGA"
set_instance_assignment -name IO_STANDARD "2.5 V" -to address[10]
set_instance_assignment -name CURRENT_STRENGTH_NEW "MAXIMUM CURRENT" -to address[10]
```

When you make an assignment in the Assignment Editor or the Pin Planner, display the equivalent Tcl command in the Messages window by performing the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, select **Assignment Editor**. The **Assignment Editor** page opens.
3. Turn on **Echo Tcl Commands**.
4. Click **OK**.

For more information about using Tcl scripts to create pin-related assignments, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Chip Planner or Timing Closure Floorplan

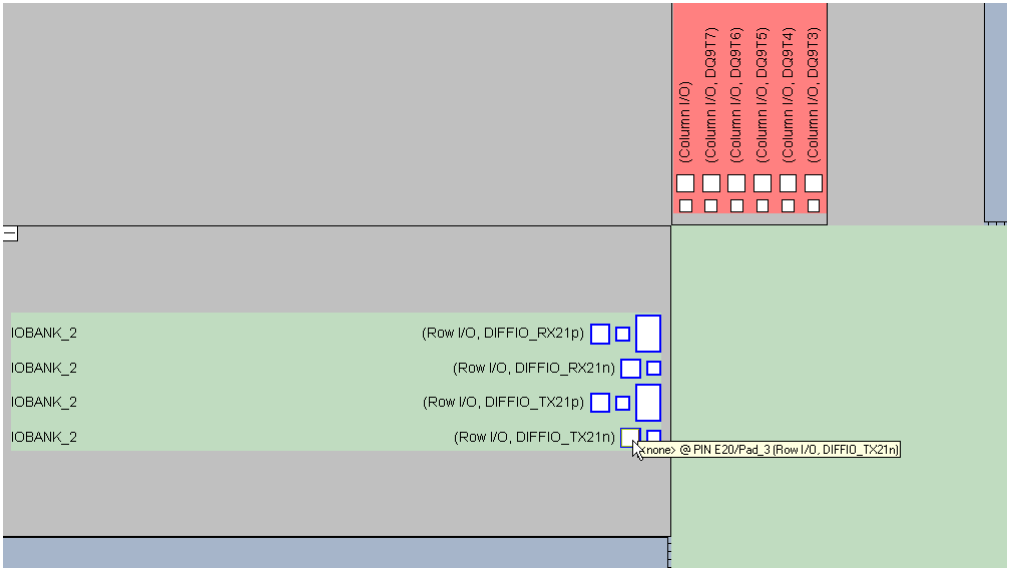
The floorplan of the device shows the pins in the same order as the pads of the device. Understanding the relative distance between a pad and related logic can help you meet your timing requirements.

You can view the floorplan of the device in the Chip Planner or the Timing Closure Floorplan. For more information about supported device families in the Chip Planner or the Timing Closure Floorplan, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Use either tool to find the distances between user I/O pads and V_{CC}, GND, and VREF pads to avoid signal integrity issues (Figure 5–41).

For more information about pin placement guidelines, refer to the *Selectable I/O Standards* chapter of the appropriate device handbook.

Figure 5–41. Timing Closure Floorplan of EP1C6F256I7



You can create a pin location assignment by selecting a pin and selecting a desired location. To do this, perform the following steps:

1. To open the Timing Closure Floorplan, on the Assignment menu, click **Timing Closure Floorplan**. To open the Chip Planner, on the Tools menu, click **Chip Planner (Floorplan & Chip Editor)**.
2. On the View Chip Planner, point to Utility Windows and click **Node Finder**. The **Node Finder** dialog box appears.
3. In the **Filter** list, select **Pins: all** and click **List** to see all the nodes in the design.
4. Select a node from the **Nodes Found** list and drag the selection into a pin location in the floorplan.



For more information about using the Timing Closure Floorplan, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Synthesis Attributes

Synthesis attributes allow you to embed assignments in your HDL code. The Quartus II software reads these synthesis attributes and translates them into assignments. The Quartus II integrated synthesis supports `chip_pin`, `useioff`, and `altera_attribute` synthesis attributes.



For more information about integrated synthesis, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

For synthesis attribute support by third-party synthesis tools, contact your vendor.

chip_pin and useioff

You can use the `chip_pin` and `useioff` synthesis attributes to embed pin location and fast output/input register assignments, respectively. For all other assignments, including pin-related assignments, use the `altera_attribute` synthesis attribute as discussed in "altera_attribute".

Synthesis attributes translated into assignments are stored in the database and take precedence over other assignments in the QSF. Example 5–2 and Example 5–3 embed a location and fast input assignment into both a Verilog HDL and VHDL design file using the `chip_pin` and `useioff` synthesis attributes.

Example 5–2. Verilog HDL Example

```
input my_pin1 /* synthesis chip_pin = "C1" useioff = 1 */;
```

Example 5–3. VHDL Example

```
entity my_entity is
      port(
                  my_pin1: in std_logic
      );
end my_entity;

architecture rtl of my_entity is
attribute useioff : boolean;
attribute useioff of my_pin1 : signal is true;
attribute chip_pin : string;
attribute chip_pin of my_pin1 : signal is "C1";
begin -- The architecture body
end rtl;
```

altera_attribute

To create other pin-related assignments, use the `altera_attribute` attribute. The `altera_attribute` attribute is understood only by the Quartus II integrated synthesis and supports all types of instance assignments. Example 5–4 and Example 5–5 use `altera_attribute` to embed the fast input register and I/O standard assignments into both a Verilog HDL and a VHDL design file.

Example 5–4. Verilog HDL Example

```
input my_pin1  /* synthesis altera_attribute = "-name FAST_INPUT_REGISTER
ON; -name IO_STANDARD \"2.5 V\" " */ ;
```

Example 5–5. VHDL Example

```
entity my_entity is
      port(
            my_pin1: in std_logic
      );
end my_entity;
architecture rtl of my_entity is
begin

attribute altera_attribute : string;
attribute altera_attribute of my_pin1: signal is "-name FAST_INPUT_REGISTER
ON;
-- The architecture body
end rtl;
```



For detailed information about synthesis attributes and their usage syntax, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Validating Pin Assignments

You must validate all pin-related assignments in your design. You can enable the live I/O check feature and use I/O Assignment Analysis to validate pin-related assignments against the predefined I/O rules encoded in the Quartus II software. To fully validate these assignments against all the I/O timing checks, you must perform full compilation.

Using the Live I/O Check Feature to Validate Pin Assignments

In the Quartus II software version 7.2 and later, the live I/O check feature provides live I/O rules checking capability. When the live I/O check feature is enabled, pin-related assignment error and warning messages appear immediately in the Quartus II messages window as you create pin-related assignments in the Pin Planner.

The Quartus II software includes built-in I/O rules to guide you in pin placement. The Quartus II software checks your pin-related assignments against these rules at various stages of the design flow. With the live I/O check feature, your pin-related assignments are checked immediately against a subset of these rules. This feature enhances your productivity by showing you warnings and errors as you create pin-related assignments, before you proceed to the next step in your design flow.

The most basic I/O rules are the I/O buffer rules. The I/O buffer rules checked by the live I/O Check feature include:

- VCCIO voltage compatibility rules
- VREF Voltage compatibility rules

- Electromigration (current density rules)
- Simultaneous Switching Output (SSO) rules
- I/O properties compatibility rules such as drive strength compatibility, I/O standard compatibility, PCI_IO clamp diode compatibility, I/O direction compatibility

An additional category of I/O rules is the set of I/O system rules. These rules can be checked only after you generate a synthesized (mappped) netlist of your design. The I/O system rules are checked when you perform I/O assignment analysis as described in "Using I/O Assignment Analysis to Validate Pin Assignments" on page 5–67.

You can enable or disable the live I/O check feature at any time. By default, the live I/O check feature is turned off.

To enable or disable the live I/O check feature in the Quartus II user interface:

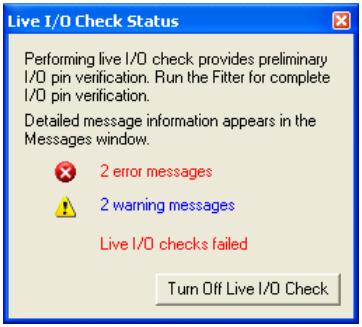
1. Verify the Pin Planner tool in the Quartus II software is active.
2. In the Quartus II Processing menu, select **Enable Live I/O Check,** or, in the Pin Planner, click on the **Live I/O Check** icon.

While the live I/O check feature is enabled, the Quartus II software immediately checks whether your new pin-related assignments and revisions pass the basic I/O buffer rules. The detailed messages are printed in the messages window of the Quartus II software. The Live I/O Check Status window displays the total numbers of errors and warnings while you create and edit pin-related assignments. To open the Live I/O Check Status window, shown in Figure 5–42, in the Quartus II View menu, click on **Live I/O Check Status**.



For details about a specific message, refer to the Quartus II Help.

Figure 5–42. Live I/O Check Status Window in the Quartus II Software



Though the live I/O check feature checks all the basic I/O buffer rules, you must use I/O assignment analysis to validate your pin-related assignments against the complete set of I/O system rules. All rules including the basic I/O buffer rules and I/O system rules can be found in Table 5–5 on page 5–83 and Table 5–6 on page 5–84.

Using I/O Assignment Analysis to Validate Pin Assignments

This section describes a design flow that includes making and analyzing pin assignments with the **Start I/O Assignment Analysis** command in the Quartus II software during and after the development of your HDL design.

The **Start I/O Assignment Analysis** command allows you to check your I/O assignments early in the design process. With this command, you can check the legality of pin assignments before, during, or after you compile your design. If design files are available, you can use this command to perform more thorough legality checks on your design's I/O pins and surrounding logic. These checks include proper reference voltage pin usage, valid pin location assignments, and acceptable mixed I/O standards.



The **Start I/O Assignment Analysis** command can be used for designs that target Stratix series, Cyclone® series, and MAX® II device families.

I/O Assignment Analysis Design Flows

The I/O assignment analysis design flows depend on whether your project contains design files. The following examples show two different circumstances in which I/O assignment analysis can be used:

- Use the flow shown in Figure 5–43 on page 5–69 if the board layout must be complete before starting the FPGA design. This flow does not require design files and checks the legality of your pin assignments.
- With a complete design, use the flow shown in Figure 5–45 on page 5–72. This flow thoroughly checks the legality of your pin assignments against any design files provided.

Each flow involves creating pin assignments, running the analysis, and reviewing the report file.

You should run the analysis each time you add or modify a pin-related assignment. You can use the **Start I/O Assignment Analysis** command frequently because it completes quickly.

The analysis checks pin assignments and surrounding logic for illegal assignments and violations of board layout rules. For example, the analysis checks whether your pin location supports the assigned I/O standard, current strength, supported V_{REF} voltages, and whether a PCI diode is permitted.

Along with the pin-related assignments, the **Start I/O Assignment Analysis** command also checks blocks that directly feed or are fed by resources such as a PLLs, LVDS, or gigabit transceiver blocks.

I/O Assignment Analysis without Design Files

During the early stages of developing an FPGA device, board layout engineers may request preliminary or final pin-outs. It is time consuming to manually check whether the pin-outs violate any design rules. Instead, use the **Start I/O Assignment Analysis** command to quickly perform basic checks on the legality of your pin assignments.

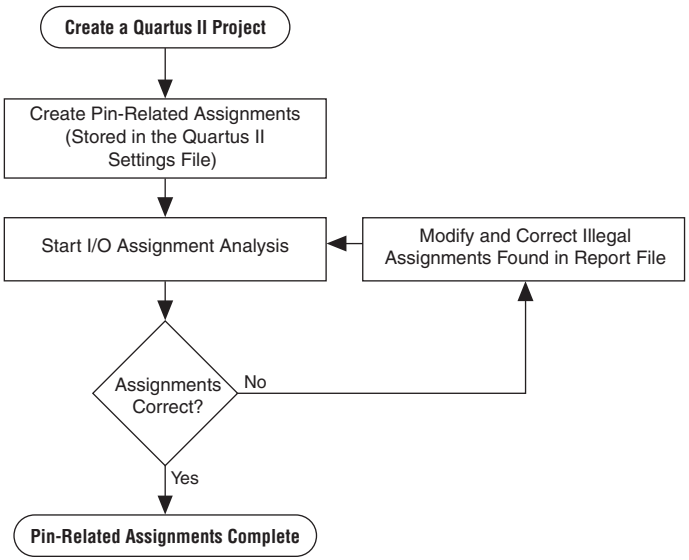


Without a complete design, the analysis performs limited checks and cannot guarantee that your assignments do not violate design rules.

The I/O Assignment Analysis command can perform limited checks on pin assignments made in a Quartus II project that has a device specified, but may not yet include any HDL design files. For example, you can create a Quartus II project with only a target device specified and create pin-related assignments based on circuit board layout considerations that

are already determined. Even though the Quartus II project does not yet contain any design files, you can reserve input and output pins and make pin-related assignments for each pin using the Pin Planner or Assignment Editor. After you assign an I/O standard to each reserved pin, run the I/O Assignment Analysis to ensure that there are no I/O standard conflicts in each I/O bank. Figure 5–43 shows the work flow for assigning and analyzing pin-outs without design files.

Figure 5–43. Assigning and Analyzing Pin-Outs without Design Files



To assign and analyze pin-outs using the **Start I/O Assignment Analysis** command without design files, perform the following steps:

1. In the Quartus II software, create a project.

2. Use the Pin Planner, Assignment Editor, or a Tcl script to create pin locations and related assignments. For the I/O assignment analysis to determine the type of pin, you must reserve your I/O pins. For information about reserving pins in the Pin Planner, refer to "Creating Reserved Pin Assignments" on page 5–38. For information about reserving pins in the Assignment Editor, refer to "Reserving Pins" on page 5–75.

   If you make pin-related assignments in the Mentor Graphics I/O Designer software, you can import an FPGA Xchange file into the Quartus II software.

3. To start the analysis, on the Processing menu, point to Start and click **Start I/O Assignment Analysis**.

   For information about using a Tcl script or command prompt to start the analysis, refer to "Scripting Support" on page 5–85.

4. View the messages in the Compilation Report window, Fitter report file (<*project name*>.**fit.rpt**), or in the Messages window.

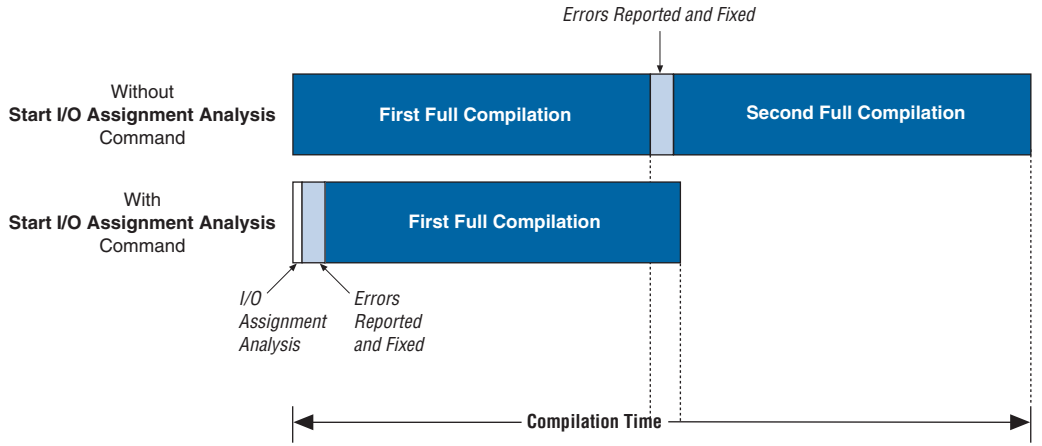
5. Correct any errors and violations reported by the I/O assignment analysis.

Repeat steps 1 through 5 until all of the errors are corrected.

I/O Assignment Analysis with Design Files

During a full compilation, the Quartus II software does not report illegal pin assignments until the Fitter stage. To validate pin assignments sooner, run the **Start I/O Assignment Analysis** command after performing analysis and synthesis and before performing a full compilation. Typically, the analysis runs quickly. Figure 5–44 shows the benefits of using the **Start I/O Assignment Analysis** command.

Figure 5–44. Saving Compilation Time with the Start I/O Assignment Analysis Command



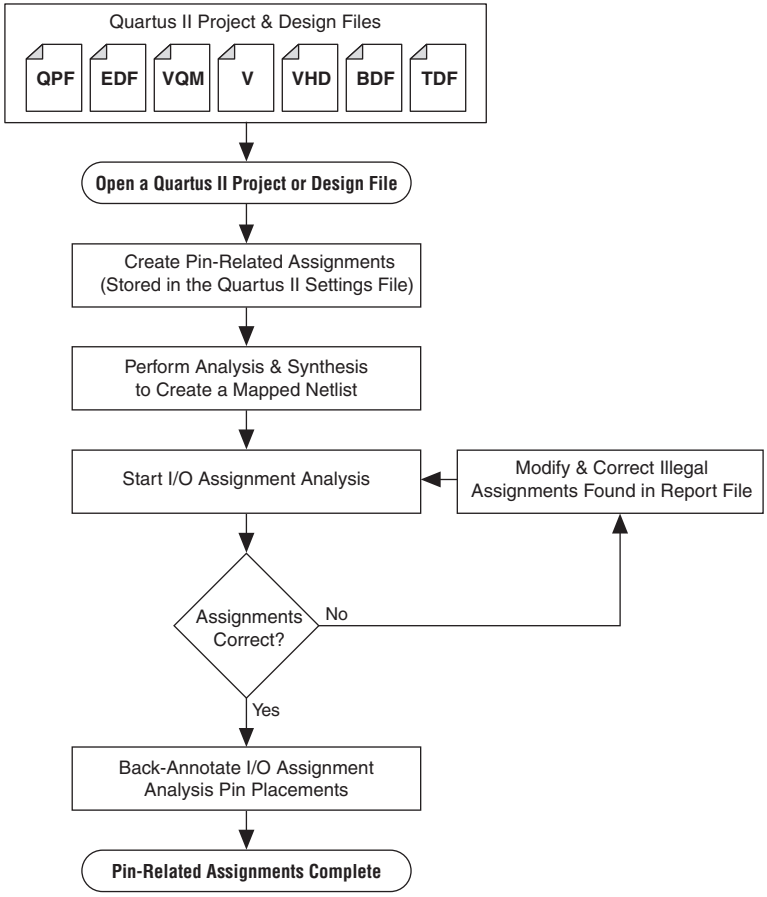
The rules that are checked by the I/O assignment analysis depend on the completeness of the design. With a complete design, the **Start I/O Assignment Analysis** command thoroughly checks the legality of all pin-related assignments. With a partial design, which can be just the

top-level wrapper file, the **Start I/O Assignment Analysis** command checks the legality of those pin-related assignments for which it has enough information.

For example, you might assign a clock to a user I/O pin instead of assigning it to a dedicated clock pin, or design the clock to drive a PLL that has not yet been instantiated in the design. Because the **Start I/O Assignment Analysis** command does not account for the logic that the pin drives, it is not able to check that only a dedicated clock input pin can drive the clock port of a PLL.

To obtain better coverage, analyze as much of the design as possible, especially logic that connects to pins. For example, if your design includes PLLs or LVDS blocks, you should include these MegaWizard Plug-In Manager-generated files in your project for analysis (Figure 5–45).

Figure 5–45. Assigning and Analyzing Pin-Outs with Design Files



To assign and analyze pin-outs using the **Start I/O Assignment Analysis** command with design files, perform the following steps:

1. Create a project including your design files.
2. Create pin-related assignments with the Pin Planner or Assignment Editor.

   You can also create pin-related assignments by importing them from a CSV or FPGA Xchange file, executing Tcl commands, or editing the QSF directly. On the Processing menu, point to Start and click **Start Analysis & Synthesis** to generate an internal mapped netlist.

   For information about using a Tcl script or the command prompt to start the analysis, refer to "Scripting Support" on page 5–85.
3. On the Processing menu, point to Start and click **Start I/O Assignment Analysis** to start the analysis.
4. View the messages in the Compilation Report or in the Messages window.
5. Use the Pin Planner or Assignment Editor to correct any errors and violations reported.
6. Use the **Start I/O Assignment Analysis** command until all errors are corrected.

Using Output Enable Group Logic Option Assignments with I/O Assignment Analysis

Each device has a certain number of VREF pins, and each VREF pin supports a certain number of I/O pins. Check the device pin-outs to locate the VREF pins and their associated I/O pins. A VREF pin and its supported I/O pins are called a VREF bank. The VREF pins are only used for VREF I/O standards; for example, SSTL and HSTL input pins. VREF outputs do not require the VREF pin. When a voltage-referenced input is present in a VREF bank, only a certain number of outputs can be present in that VREF bank. For the Stratix II flip chip package, only 20 outputs can be present in a VREF bank when a VREF I/O standard input is present in that bank.

For interfaces that use bidirectional VREF I/O pins, the VREF restriction must be met when the pins are driving in either direction. If a set of bidirectional signals are controlled by different output enables, the **I/O Assignment Analysis** command treats these as independent output

enables. Use the output enable group logic option assignment to treat the set of bidirectional signals as a single output enable. This is important in the case of external memory interfaces.

For example, in the case of a DDR2 interface in a Stratix II device, a Stratix II device can have 30 pins in a VREF group. Each byte lane for a ×8 DDR2 interfaces has 1 DQS pin and 8 DQ pins, for a total of 9 pins per byte lane. DDR2 uses SSTL18 as its I/O standard, which is a VREF I/O standard. In typical interfaces, each byte lane has its own output enable. In this example, the DDR2 interface has 4 byte lanes. Using 30 I/O pins in a VREF group, there are 3 byte lanes, and an extra byte lane that supports the 3 remaining pins. If you do not use the output enable group logic option assignment, the I/O Assignment Analysis command analyzes each byte lane as an independent group driven by a unique output enable. With this arrangement, the worst-case scenario is when the 3 pins are inputs, and the other 27 pins are outputs. In this case, the 27 output pins violate the 20-output pin limit.

In a DDR2 interface, all DQS and DQ pins are always driven in the same direction. Therefore, the I/O Assignment Analysis reports an error that is not applicable to your design. Assigning an output enable group logic option assignment to the DQS and DQ pins forces the I/O Assignment Analyzer to check these pins as a group driven by a common output enable. When using the output enable group logic option assignment, the DQS and DQ pins are checked as all input pins or all output pins. This does not violate the rules described in Table 5–5 on page 5–83 and Table 5–6 on page 5–84.

The value for the output enable group logic option assignment should be an integer value. All sets of signals that are driving in the same direction should be given the same integer value. The output enable group logic option assignment can also be used with pins that are driven only at certain times. For example, the data mask signal in DDR2 interfaces is an output signal, but it is driven only when the DDR2 is writing (bidirectional signals are outputs). Therefore, an output enable group logic option assignment should assign to the data mask the same value as to the DQ and DQS signals.

Output enable groups can also be used on VREF input pins. If the VREF input pins are not active during the time the outputs are driving, add the VREF input pins to the output enable group. This procedure removes the VREF input pins from the VREF analysis. For example, the QVLD signal for RLDRAM II is only active during a read. During a write, the QVLD pin is not active and so it does not count as an active VREF input pin within the VREF group. The QVLD pins can be placed in the same output enable group as the RLDRAM II data pins.

Inputs for I/O Assignment Analysis

The **Start I/O Assignment Analysis** command reads the following inputs:

- Internal mapped netlist
- Quartus II Settings File (QSF)

The internal mapped netlist is used when you have a partial or complete design. The QSF is always used to read all pin-related assignments for analysis.

Generating a Mapped Netlist

The **Start I/O Assignment Analysis** command uses a mapped netlist, if available, to identify the pin type and the surrounding logic. The mapped netlist is stored internally in the Quartus II software database.

To generate a mapped netlist, on the Processing menu, point to Start and click **Start Analysis & Synthesis**.

To use the `quartus_map` executable to run analysis and synthesis, type the following command at a system command prompt:

```
quartus_map <project name> ↵
```

Creating Pin-Related Assignments

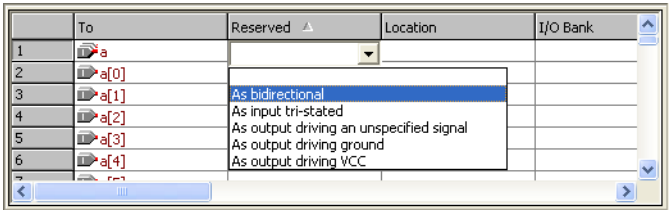
The **I/O Assignment Analysis** command reads a QSF containing all of your pin-related assignments. These pin-related assignments include pin settings such as I/O standards, drive strength, and location assignments. The following sections highlight some of the location assignments you can make.

Reserving Pins

If you do not have any design files, you can still reserve pin locations and create pin-related assignments. Reserving pins is necessary so that the **Start I/O Assignment Analysis** command has information about the pin and the pin type (input, output, or bidirectional) to correctly analyze the pins.

To reserve a pin, on the Assignments menu, click **Assignment Editor**. In the **Category** list, click **Pin** to open the Pin assignment category. Double-click the cell in the **Reserved** column that corresponds to the pin that you want to reserve. Use the drop-down arrow to select from the reserve pin options (Figure 5–46).

Figure 5–46. Reserving an Input Pin with the Assignment Editor



For more information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

You can also reserve pins using the Pin Planner. For more information about the Pin Planner, refer to "Creating Reserved Pin Assignments" on page 5–38.

Location Assignments

You can create the following types of location assignments for your design and its reserved pins:

- Pin number
- I/O bank
- VREF group
- Edge

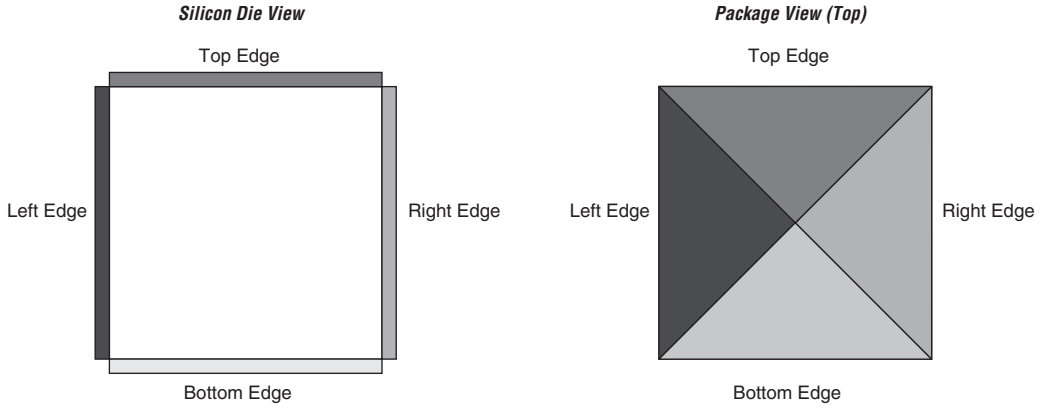
I/O bank, VREF group, and Edge location assignments are supported only for Stratix and Cyclone series device families.

You can assign a location to your pins using the Pin Planner or the Assignment Editor. To make a pin location assignment using the Assignment Editor, on the Assignments menu, click **Assignment Editor** and select the **Pin** category from the **Category** list. Type the pin name and select a location from the **Location** list.

It is common to place a group of pins (or bus) with compatible I/O standards in the same I/O bank or VREF group. For example, two buses with two compatible I/O standards, such as 2.5 V and SSTL-II, can be placed in the same I/O bank.

An easy way to place large buses that exceed the pins available in a particular I/O bank is to use edge location assignments. Edge location assignments improve the circuit board routing ability of large buses, because they are close together near an edge. Figure 5–47 shows the Altera device package edges.

Figure 5–47. Die View and Package View of the Four Edges on an Altera Device



Suggested and Partial Placement

The **Start I/O Assignment Analysis** command automatically assigns suggested pin locations to unassigned pins in your design so it can perform pin legality checks. For example, if you assign an edge location to a group of LVDS pins, the **I/O Assignment Analysis** command assigns pin locations for each LVDS pin in the specified edge location and then performs legality checks.

To accept these suggested pin locations, on the Assignments menu, click **Back-Annotate Assignments**, select **Pin & device** assignments, and click **OK**. Back-annotation saves your pin and device assignments in the QSF.

Understanding the I/O Assignment Analysis Report and Messages

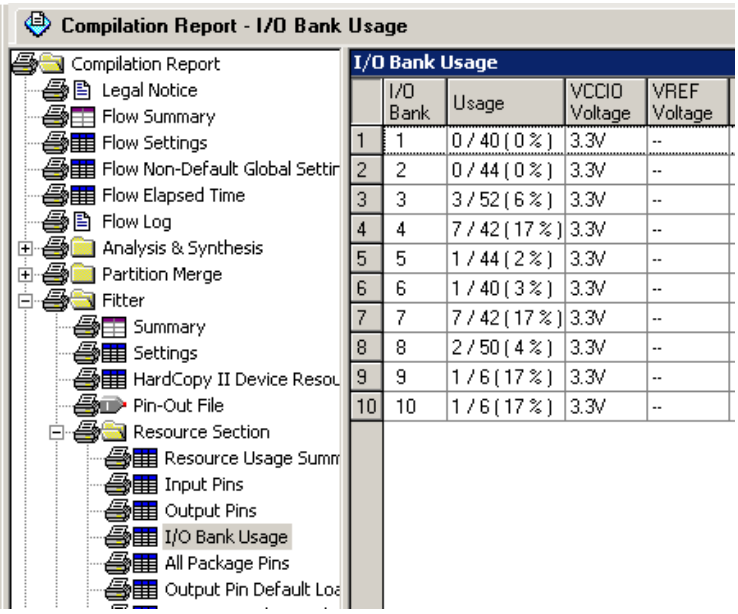
The **Start I/O Assignment Analysis** command generates detailed analysis reports and a PIN file. The detailed messages in the reports help you quickly understand and resolve pin assignment errors. Each message includes a related node name and a description of the problem.

To view the report file, on the Project menu, click **Compilation Report.** The Fitter section of the Compilation Report contains the following sections:

- Summary
- Settings
- Resource Section
- I/O Rules Section
- Device Options
- Advanced Fitter Data
- Pin-Out File
- Fitter Messages

The Resource Section categorizes the pins as Input Pins, Output Pins, and Bidir Pins. View the utilization of each I/O bank in your device in the I/O Bank Usage section (Figure 5–48).

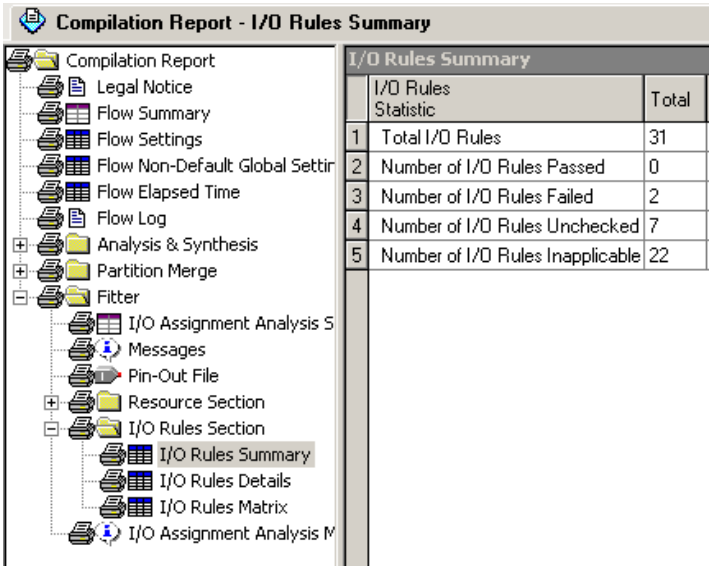
Figure 5–48. I/O Bank Usage Summary in the I/O Assignment Analysis Report



	I/O Bank	Usage	VCCIO Voltage	VREF Voltage
1	1	0 / 40 (0 %)	3.3V	--
2	2	0 / 44 (0 %)	3.3V	--
3	3	3 / 52 (6 %)	3.3V	--
4	4	7 / 42 (17 %)	3.3V	--
5	5	1 / 44 (2 %)	3.3V	--
6	6	1 / 40 (3 %)	3.3V	--
7	7	7 / 42 (17 %)	3.3V	--
8	8	2 / 50 (4 %)	3.3V	--
9	9	1 / 6 (17 %)	3.3V	--
10	10	1 / 6 (17 %)	3.3V	--

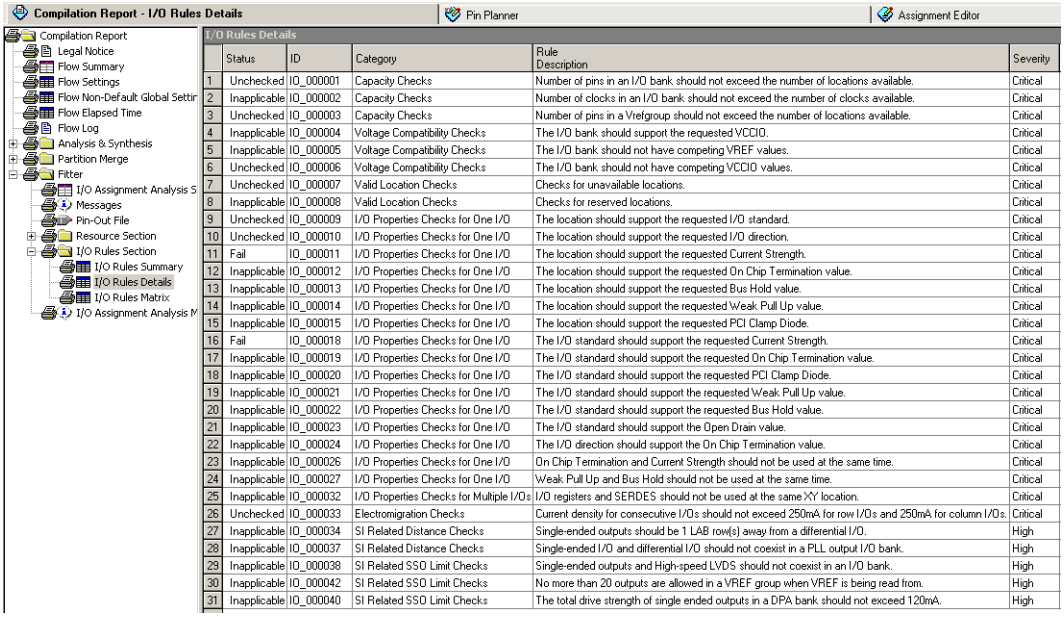
The I/O Rules Section includes detailed information about the I/O rules tested during I/O Assignment Analysis, in three sub-reports. The I/O Rules Summary report provides a quick summary of the number of I/O rules tested and how many applicable rules passed, how many failed, and how many were unchecked because of other failing rules (Figure 5–49).

Figure 5–49. I/O Rules Summary Report



The I/O Rules Details report provides detailed information on all I/O rules. Applicable rules indicate whether they passed, failed, or could not be checked (Figure 5–50). All rules are given a level of severity from Low to Critical to indicate their level of importance for an effective analysis.

Figure 5–50. I/O Rules Details Report



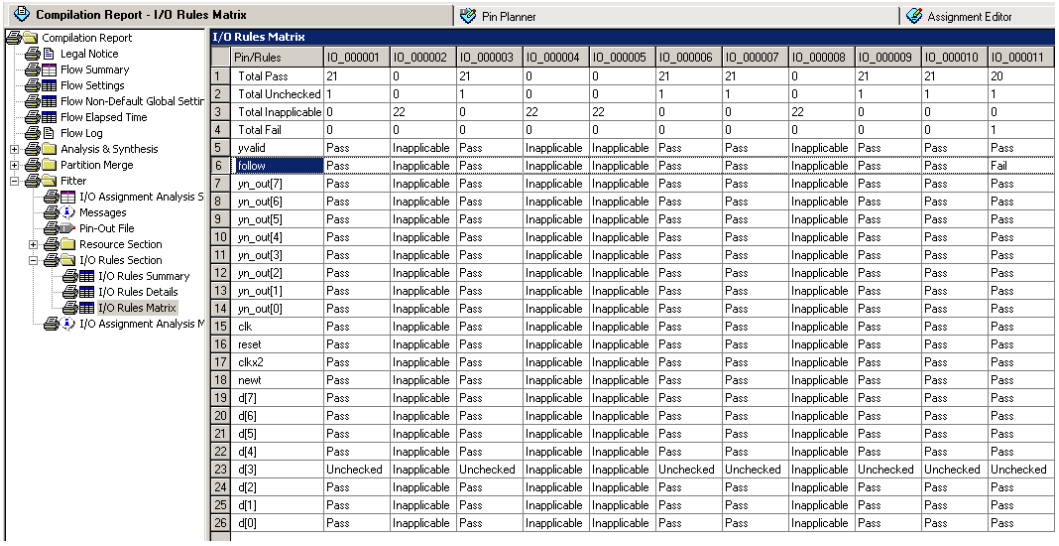
I/O Rules Details

	Status	ID	Category	Rule Description	Severity
1	Unchecked	IO_000001	Capacity Checks	Number of pins in an I/O bank should not exceed the number of locations available.	Critical
2	Inapplicable	IO_000002	Capacity Checks	Number of clocks in an I/O bank should not exceed the number of clocks available.	Critical
3	Unchecked	IO_000003	Capacity Checks	Number of pins in a Vrefgroup should not exceed the number of locations available.	Critical
4	Inapplicable	IO_000004	Voltage Compatibility Checks	The I/O bank should support the requested VCCIO.	Critical
5	Inapplicable	IO_000005	Voltage Compatibility Checks	The I/O bank should not have competing VREF values.	Critical
6	Unchecked	IO_000006	Voltage Compatibility Checks	The I/O bank should not have competing VCCIO values.	Critical
7	Unchecked	IO_000007	Valid Location Checks	Checks for unavailable locations.	Critical
8	Inapplicable	IO_000008	Valid Location Checks	Checks for reserved locations.	Critical
9	Unchecked	IO_000009	I/O Properties Checks for One I/O	The location should support the requested I/O standard.	Critical
10	Unchecked	IO_000010	I/O Properties Checks for One I/O	The location should support the requested I/O direction.	Critical
11	Fail	IO_000011	I/O Properties Checks for One I/O	The location should support the requested Current Strength.	Critical
12	Inapplicable	IO_000012	I/O Properties Checks for One I/O	The location should support the requested On Chip Termination value.	Critical
13	Inapplicable	IO_000013	I/O Properties Checks for One I/O	The location should support the requested Bus Hold value.	Critical
14	Inapplicable	IO_000014	I/O Properties Checks for One I/O	The location should support the requested Weak Pull Up value.	Critical
15	Inapplicable	IO_000015	I/O Properties Checks for One I/O	The location should support the requested PCI Clamp Diode.	Critical
16	Fail	IO_000018	I/O Properties Checks for One I/O	The I/O standard should support the requested Current Strength.	Critical
17	Inapplicable	IO_000019	I/O Properties Checks for One I/O	The I/O standard should support the requested On Chip Termination value.	Critical
18	Inapplicable	IO_000020	I/O Properties Checks for One I/O	The I/O standard should support the requested PCI Clamp Diode.	Critical
19	Inapplicable	IO_000021	I/O Properties Checks for One I/O	The I/O standard should support the requested Weak Pull Up value.	Critical
20	Inapplicable	IO_000022	I/O Properties Checks for One I/O	The I/O standard should support the requested Bus Hold value.	Critical
21	Inapplicable	IO_000023	I/O Properties Checks for One I/O	The I/O standard should support the Open Drain value.	Critical
22	Inapplicable	IO_000024	I/O Properties Checks for One I/O	The I/O direction should support the On Chip Termination value.	Critical
23	Inapplicable	IO_000026	I/O Properties Checks for One I/O	On Chip Termination and Current Strength should not be used at the same time.	Critical
24	Inapplicable	IO_000027	I/O Properties Checks for One I/O	Weak Pull Up and Bus Hold should not be used at the same time.	Critical
25	Inapplicable	IO_000032	I/O Properties Checks for Multiple I/Os	I/O registers and SERDES should not be used at the same XY location.	Critical
26	Unchecked	IO_000033	Electromigration Checks	Current density for consecutive I/Os should not exceed 250mA for row I/Os and 250mA for column I/Os.	Critical
27	Inapplicable	IO_000034	SI Related Distance Checks	Single-ended outputs should be 1 LAB row(s) away from a differential I/O.	High
28	Inapplicable	IO_000037	SI Related Distance Checks	Single-ended I/O and differential I/O should not coexist in a PLL output I/O bank.	High
29	Inapplicable	IO_000038	SI Related SSO Limit Checks	Single-ended outputs and High-speed LVDS should not coexist in an I/O bank.	High
30	Inapplicable	IO_000042	SI Related SSO Limit Checks	No more than 20 outputs are allowed in a VREF group when VREF is being read from.	High
31	Inapplicable	IO_000040	SI Related SSO Limit Checks	The total drive strength of single ended outputs in a DPA bank should not exceed 120mA.	High

The I/O Rules Matrix shows how each I/O rule was tested on each pin in the design (Figure 5–51). Applicable rules that could be checked either pass or fail for each pin.

To find and make pin assignment adjustments on a pin that fails an I/O rule, right-click the pin name. Point to Locate, and select a location where the pin exists, such as the Pin Planner. Make appropriate changes to fix the pin assignments and rerun I/O Assignment Analysis. Check the resulting I/O Rules Matrix to verify that your changes fixed the problem and allowed the failing pin assignment to pass. To rerun I/O rule analysis, on the Processing menu, point to Start and click **Start I/O Assignment Analysis**.

Figure 5–51. I/O Rules Matrix



I/O Rules Matrix

	Pin/Rules	IO_000001	IO_000002	IO_000003	IO_000004	IO_000005	IO_000006	IO_000007	IO_000008	IO_000009	IO_000010	IO_000011
1	Total Pass	21	0	21	0	0	21	21	0	21	21	20
2	Total Unchecked	1	0	1	0	0	1	1	0	1	1	1
3	Total Inapplicable	0	22	0	22	22	0	0	22	0	0	0
4	Total Fail	0	0	0	0	0	0	0	0	0	0	1
5	yvalid	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
6	follow	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Fail
7	yn_out[7]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
8	yn_out[6]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
9	yn_out[5]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
10	yn_out[4]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
11	yn_out[3]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
12	yn_out[2]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
13	yn_out[1]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
14	yn_out[0]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
15	clk	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
16	reset	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
17	clkx2	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
18	newt	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
19	d[7]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
20	d[6]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
21	d[5]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
22	d[4]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
23	d[3]	Unchecked	Inapplicable	Unchecked	Inapplicable	Inapplicable	Unchecked	Unchecked	Inapplicable	Unchecked	Unchecked	Unchecked
24	d[2]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
25	d[1]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
26	d[0]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass

The **Fitter Messages** page stores all messages including errors, warnings, and information messages.

You can view the detailed messages in the **Fitter Messages** page in the compilation report and in the **Processing** tab in the Messages window. To open the Messages window, on the View menu, point to Utility windows and click Messages.

Use the **Location** box to help resolve the error messages. Select from the **Location** list, and click **Locate**.

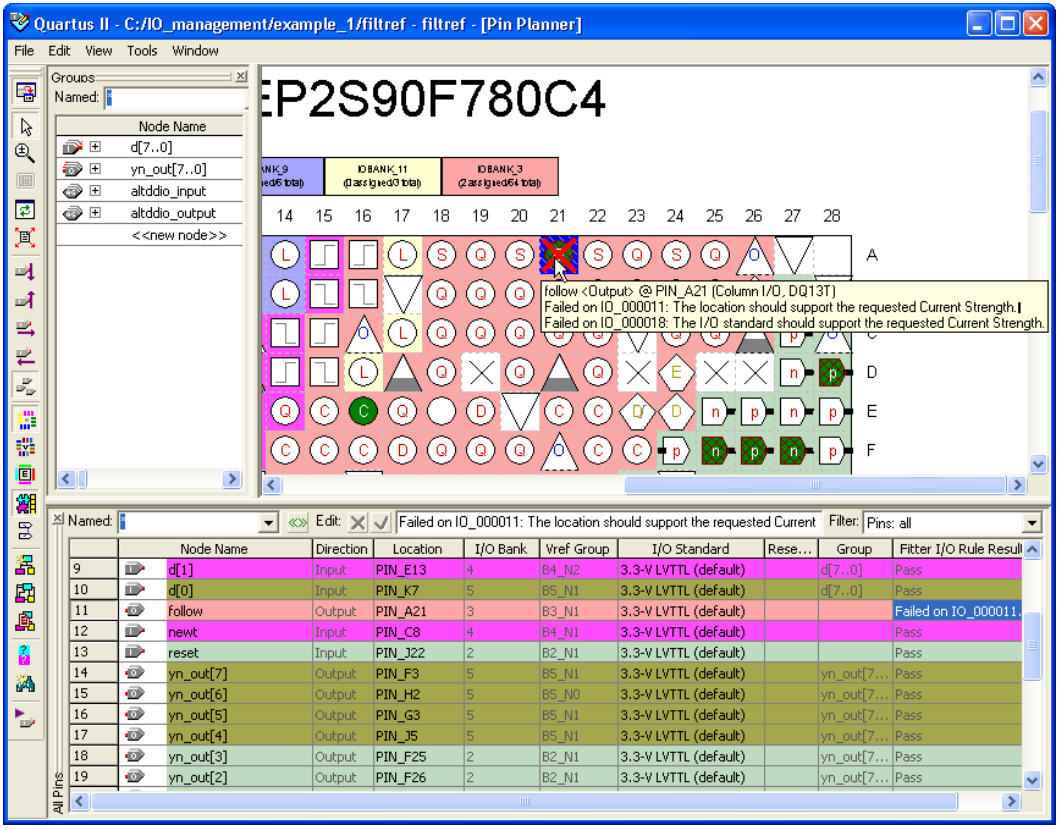
Figure 5–52 shows an example of error messages reported by I/O assignment analysis.

Figure 5–52. Error Message Report by I/O Assignment Analysis

```
Error: Current Strength logic option is set to 10mA for pin follow, but setting  is not supported by I/O standard 3.3-V LVTTL
Error: Pin AF3 does not support I/O standard 3.3-V LVTTL with Current Strength 10mA for location follow
Error: Can't fit design in device
Error: Quartus II I/O Assignment Analysis was unsuccessful. 3 errors, 1 warning
```

The Fitter messages can also be seen in the Package View (Figure 5–53). Right click in the Package View and click **Show Fitter Placements** to see the failing pins. The failing pins are shown with a cross sign as shown in Figure 5–53 and a tool tip is displayed when the mouse cursor is pointed over the failing pin. The tooltip displays the failed I/O rules.

Figure 5–53. Tool Tip



You can correct the I/O assignment analysis failure shown for the pin in Figure 5–52 and Figure 5–53 easily by setting the proper current drive strength for the I/O standard assigned for that pin. Current drive strength can be set in Assignment editor using the "Current Drive Strength" Assignment.

For more information about the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

The effectiveness of the I/O Assignment Analysis is relative to the completeness of your pin-related assignments and design. To ensure your design functions correctly, include all pin-related assignments and as many design files as possible in your Quartus II project.

Table 5–5 on page 5–83 and Table 5–6 on page 5–84 list a subset of the I/O rule checks performed when you run an I/O Assignment Analysis with and without design files.

For more detailed information about each I/O rule, refer to the appropriate device handbook.

Table 5–5. Examples of I/O Rule Checks (Part 1 of 2) *Note (1)*

Rule	Description	Device Families	HDL Required?
I/O bank capacity	Checks the number of pins assigned to an I/O bank against the number of pins allowed in the I/O bank.	All	No
I/O bank V_{CCIO} voltage compatibility	Checks that no more than one `VCCIO` is required for the pins assigned to the I/O bank.	All	No
I/O bank V_{REF} voltage compatibility	Checks that no more than one VREF is required for the pins assigned to the I/O bank.	All	No
I/O standard and location conflicts	Checks whether the pin location supports the assigned I/O standard.	All	No
I/O standard and signal direction conflicts	Checks whether the pin location supports the assigned I/O standard and direction. For example, certain I/O standards on a particular pin location can only support output pins.	All	No
Differential I/O standards cannot have open drain turned on	Checks that open drain is turned off for all pins with a differential I/O standard.	All	No
I/O standard and drive strength conflicts	Checks whether the drive strength assignments are within the specifications of the I/O standard.	All	No
Drive strength and location conflicts	Checks whether the pin location supports the assigned drive strength.	All	No
`BUSHOLD` and location conflicts	Checks whether the pin location supports `BUSHOLD`. For example, dedicated clock pins do not support `BUSHOLD`.	All	No

Table 5–5. Examples of I/O Rule Checks (Part 2 of 2) *Note (1)*

Rule	Description	Device Families	HDL Required?
WEAK_PULLUP and location conflicts	Checks whether the pin location supports WEAK_PULLUP (for example, dedicated clock pins do not support WEAK_PULLUP)	All	No
Electromigration check	Checks whether combined drive strength of consecutive pads exceeds a certain limit. For example, the total current drive for 10 consecutive pads on a Stratix II device cannot exceed 200 mA.	All	No
PCI_IO clamp diode, location, and I/O standard conflicts	Checks whether the pin location along with the I/O standard assigned supports PCI_IO clamp diode.	All	No
SERDES and I/O pin location compatibility check	Checks that all pins connected to a SERDES in your design are assigned to dedicated SERDES pin locations.	All	Yes
PLL and I/O pin location compatibility check	Checks whether pins connected to a PLL are assigned to the dedicated PLL pin locations.	All	Yes

Note to Table 5–5:

(1) The supported device families are: Arria™ GX, Stratix III, Stratix II, Stratix II GX, Stratix, Stratix GX, Cyclone® III, Cyclone II, Cyclone, HardCopy, and MAX II devices.

Table 5–6. SSN-Related Rules (Part 1 of 2)

Rule	Description	Device Families (1)	HDL Required?
I/O bank can not have single-ended I/O when DPA exists	Checks that no single-ended I/O pin exists in the same I/O bank as a DPA.	Stratix II Stratix GX	No
A PLL I/O bank does not support both a single-ended I/O and a differential signal simultaneously	Checks that there are no single-ended I/O pins present in the PLL I/O Bank when a differential signal exists.	Stratix II	No
Single-ended output is required to be a certain distance away from a differential I/O pin	Checks whether single-ended output pins are a certain distance away from a differential I/O pin.	All	No
Single-ended output has to be a certain distance away from a VREF pad	Checks whether single-ended output pins are a certain distance away from a VREF pad.	Cyclone II Cyclone	No
Single-ended input is required to be a certain distance away from a differential I/O pin	Checks whether single-ended input pins are a certain distance away from a differential I/O pin.	Cyclone II Cyclone	No

Table 5–6. SSN-Related Rules (Part 2 of 2)

Rule	Description	Device Families (1)	HDL Required?
Too many outputs or bidirectional pins in a VREFGROUP when a VREF is used	Checks that there are no more than a certain number of outputs or bidirectional pins in a VREFGROUP when a VREF is used.	All	No
Too many outputs in a VREFGROUP	Checks whether too many outputs are in a VREFGROUP.	All	No

Note to Table 5–6:
(1) "All" includes the following device families: Arria GX, Stratix III, Stratix II, Stratix II GX, Stratix, Stratix GX, Cyclone III, Cyclone II, Cyclone, HardCopy, and MAX II devices.

Scripting Support

A Tcl script allows you to run procedures and determine settings described in this chapter. You can also run some of these procedures at a command prompt.

For detailed information about specific scripting command options and Tcl API packages, type the following command at a system command prompt to run the Quartus II Command-Line and Tcl API Help browser:

```
quartus_sh --qhelp ↵
```

For more information about Quartus II scripting support, including examples, refer to the *Tcl Scripting* and *Command-Line Scripting* chapters in volume 2 of the *Quartus II Handbook*.

Running the I/O Assignment Analysis

You can run the I/O Assignment Analysis with a Tcl command or with a command run at a command prompt. For more information about running the I/O Assignment Analysis, refer to "Understanding the I/O Assignment Analysis Report and Messages" on page 5–77.

Enter the following in a Tcl console or script:

```
execute_flow -check_ios
```

Type the following at a (non-Tcl) system command prompt:

```
quartus_fit <project-name> --check_ios ↵
```

Generating a Mapped Netlist
You can generate a mapped netlist with a Tcl command or with a command-line command. For more information about generating a mapped netlist, refer to "Generating a Mapped Netlist" on page 5–75.

Enter the following in the Tcl console or in a script:

```
execute_module -tool map
```

The `execute_module` command is in the flow package.

Type the following at a system command prompt:

```
quartus_map <project name>↵
```

Reserving Pins
Use the following Tcl command to reserve a pin. For more information about reserving pins, refer to "Reserving Pins" on page 5–75.

```
set_instance_assignment -name RESERVE_PIN <value> -to <signal name>
```

Valid values are:

- `"AS BIDIRECTIONAL"`
- `"AS INPUT TRI-STATED"`
- `"AS OUTPUT DRIVING AN UNSPECIFIED SIGNAL"`
- `"AS OUTPUT DRIVING GROUND"`
- `"AS SIGNALPROBE OUTPUT"`

Include the quotes when specifying the value.

Location Assignments
Use the following Tcl command to assign a signal to a pin or device location. For more information about location assignments, refer to "Location Assignments" on page 5–76.

```
set_location_assignment <location> -to <signal name>
```

Valid locations are pin location names, such as `PIN_A3`. The Stratix and Cyclone series of devices also support edge and I/O bank locations. Edge locations are `EDGE_BOTTOM`, `EDGE_LEFT`, `EDGE_TOP`, and `EDGE_RIGHT`. I/O bank locations include `IOBANK_1` up to `IOBANK_`*n*, in which *n* is the number of I/O banks in a particular device.

In Stratix III devices only, I/O bank names have the form IOBANK_*nx* where n is a number and *x* is the letter *A*, *B*, or *C*. Though I/O banks may share the same number with different letters, such as 1A and 1C, they are separate banks and not related to each other.

For more information, refer to the *Stratix III Device Handbook*.

Incorporating PCB Design Tools

Signal and pin assignments are initially made by the FPGA or ASIC designer, and it is up to the board designer to transfer these assignments to the symbols used in their system circuit schematics and board layout correctly. As the board design progresses, pin reassignments may be requested or required to optimize the layout. These reassignments must in turn be relayed to the FPGA designer, so that the new assignments can be validated with the I/O Assignment Analyzer and processed through an updated place-and-route of the FPGA.

The Quartus II software interacts with board layout tools by importing and exporting pin information files, including the QSF, the PIN file, and the FPGA Xchange file.

For more information about incorporating PCB design tools, refer to the *Cadence PCB Design Tools Support* and the *Mentor Graphics PCB Design Tools Support* chapters in volume 2 of the *Quartus II Handbook*.

Advanced I/O Timing

As part of I/O planning, especially with high-speed designs, you should take board-level signal integrity and timing into account. When adding an FPGA device with high-speed interfaces to a board design, the quality of the signal at the far end of the board route, as well as the propagation delay in getting there, is vital for proper system operation.

The Quartus II software provides features to take these factors into consideration, making the software "board-aware." The Quartus II software can take into account board routing and external devices to generate advanced timing reports and board simulation modeling files. The Quartus II software supports three different methods of analysis:

- I/O timing using a default or user-specified capacitive load with no signal integrity analysis (default)
- The Quartus II **Advanced I/O Timing** option utilizing a user-defined board trace model to produce enhanced timing reports from accurate, "board-aware" simulation models
- Full board routing simulation in third-party tools using Altera-provided or generated IBIS or HSPICE I/O models



I/O timing using a default or user-specified capacitive load is not supported for Stratix III and Cyclone III devices. Use the **Advanced I/O Timing** option for Stratix III and Cyclone III devices.

In the first method, timing reports created by the Quartus II TimeQuest Timing Analyzer and the Quartus II Classic Timing Analyzer measure t_{CO} to an I/O pin using a default or user-specified value for a capacitive load.

The second method, the Quartus II **Advanced I/O Timing** option, lets you configure a complete board trace model for each I/O standard or pin used in your design. With **Advanced I/O Timing** turned on, the Quartus II TimeQuest Timing Analyzer uses the results of simulations of the I/O buffer, package, and board trace model to generate more accurate I/O delays and extra reports to give insight into signal behavior at the system level. You can use these advanced timing reports as a guide to make changes to your I/O assignments and board design to improve timing and signal integrity.

This section details the first and second methods. The third method of analysis, the creation of simulation model files for use by third-party board simulation tools, is achieved with the IBIS and HSPICE Writers. The IBIS and HSPICE Writers in the Quartus II software can export accurate simulation models for use in applications such as Mentor Graphics HyperLynx and Synopsys HSPICE.

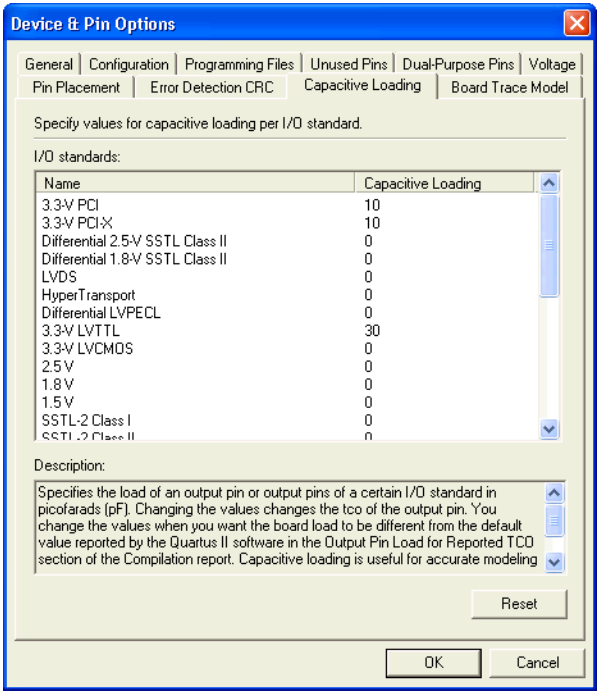


For information about creating IBIS and HSPICE models with the Quartus II software and integrating those models into HyperLynx and HSPICE simulations, refer to the *Signal Integrity Analysis with Third Party Tools* chapter in volume 3 of the *Quartus II Handbook*.

I/O Timing and Power with Capacitive Loading

When calculating t_{CO} and power for output and bidirectional pins, the Quartus II TimeQuest Timing Analyzer, the Quartus II Classic Timing Analyzer, and power analysis use a bulk capacitive load. This is the default method for these pins. You can adjust the value of the capacitive load per I/O standard to get t_{CO} and power measurements that more accurately reflect the behavior of the output or bidirectional net on your PCB. Input pins ignore this setting. To adjust the value of the capacitive load, on the Assignments menu, click **Device**. Click **Device & Pin Options**, and click the **Capacitive Loading** tab (Figure 5–54).

Figure 5–54. Capacitive Tab of the Device and Pin Options Dialog Box



All of the available I/O standards for your selected device are listed with their default loading values in picofarads (pF). Adjust the loading values as desired for the I/O standards used in your design. Power and t_{CO} measurements in the Compilation Report are adjusted based on the settings.

You can also adjust the load on any individual pin in the Groups List or All Pins List in the Pin Planner by adding the Output Pin Load column. Right-click anywhere in either list and select **Customize Columns**. Select **Output Pin Load** from the list of available custom columns, and add it to the list of visible columns. You can customize the load for individual pins or multiple pins with different I/O standards.

For more information about capacitive loading, the devices that support it, and how t_{CO} and power are adjusted based on the setting, refer to the Quartus II Help.

Enabling and Configuring Advanced I/O Timing

With the Quartus II **Advanced I/O Timing** turned on, you can expand upon the basic timing and power measurements made with the **Capacitive Loading** settings. Advanced I/O Timing gives you the ability to fully define not only the capacitive load, but also any termination components and trace impedances in the board routing for any output pin or bidirectional pin in output mode. You can configure an overall board trace model for each I/O standard as well as customize the model for specific pins using a graphical interface.

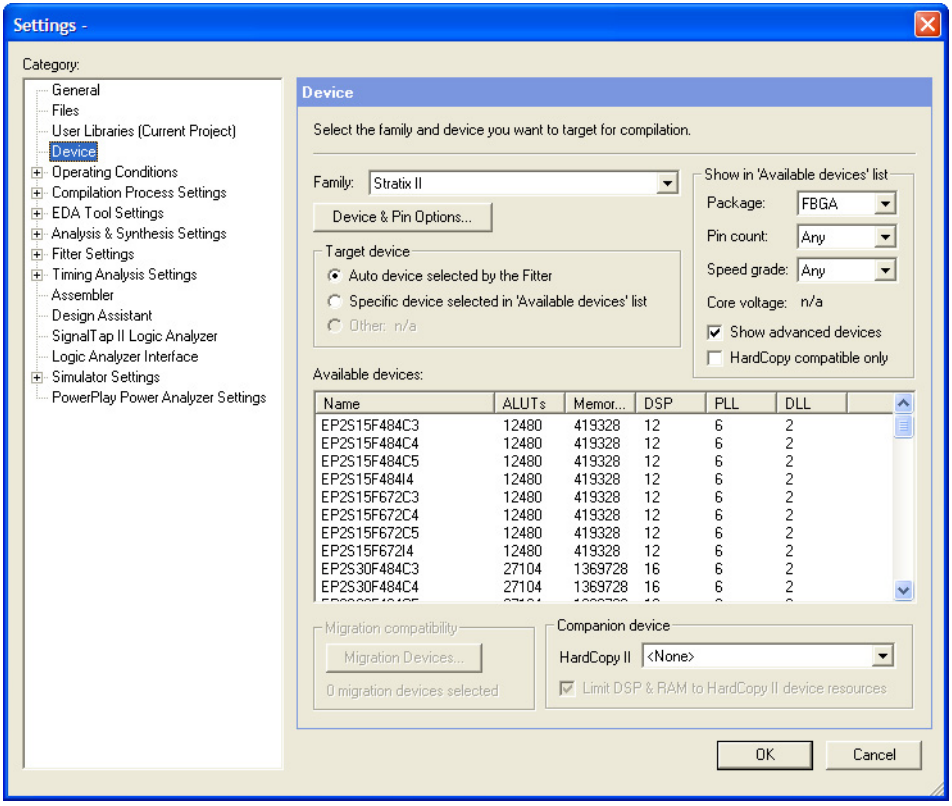
When the **Enable Advanced I/O Timing** option is turned on, the board trace model replaces the **Capacitive Loading** tab settings because the load is included in the model. For timing measurements, the entire board trace model is taken into account when calculating I/O delays. For power measurements, an effective capacitive load is used based on the sum of the capacitive elements in the model. This includes the **Near capacitance**, **Far capacitance**, and **Transmission line distributed capacitance** elements of the model.



For Stratix III and Cyclone III devices, Advanced I/O Timing is the only way to measure I/O timing. Advanced I/O Timing is currently supported for Stratix II devices also. All other devices use Capacitive Loading for I/O t_{CO} and power measurements. Check the Altera website at **www.altera.com** to determine which devices are supported in newer versions of the Quartus II software.

Before you configure a board trace model for Advanced I/O Timing, you must select a device from a supported device family for your design and you must turn on the Advanced I/O Timing option. To select a device that supports Advanced I/O Timing, on the Assignments menu, click **Device** to open the **Settings** dialog box (Figure 5–55). From the **Family** list, select the supported device. You can set the other controls under **Show in 'Available devices list'** to filter the **Available devices** list and to select any migration devices. Under the **Available devices** list, select a device. All devices in each supported family work with Advanced I/O Timing.

Figure 5–55. Device Page of the Settings Dialog Box



Turn on **Enable Advanced I/O Timing**. If the **Settings** dialog box is not currently open, on the Assignments menu, click **Settings**. In the **Category** list, click the "+" icon to expand **Timing Analysis Settings**. Select **TimeQuest Timing Analyzer**. The **TimeQuest Timing Analysis** page appears. Turn on **Enable Advanced I/O Timing**.

For Stratix III and Cyclone III devices, the **Advanced I/O Timing** option is turned on by default and is always performed when you run the Quartus II TimeQuest Timing Analyzer.

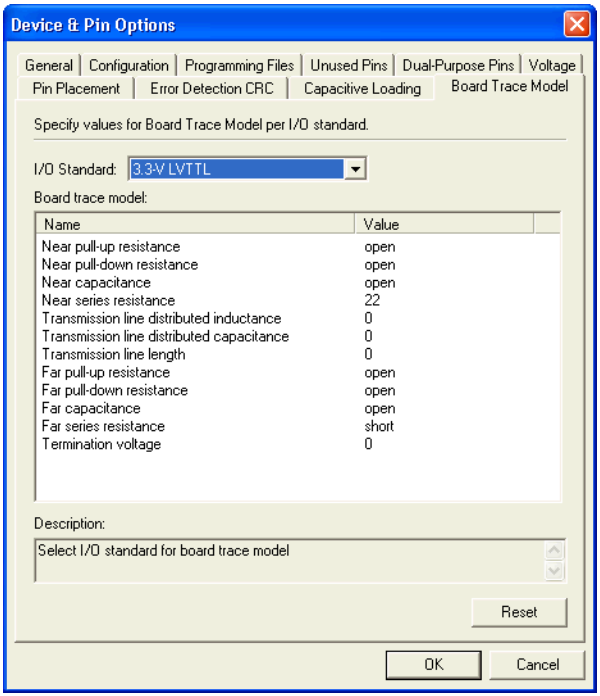
For more information on the Quartus II TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Define Overall Board Trace Models

You can now define an overall board trace model for each I/O standard in your design. This is the default model for all pins that use a particular I/O standard. After configuring the overall board trace model, customize the model for specific pins using the Board Trace Model view in the Pin Planner.

With the **Settings** dialog box open, in the **Category** list, click **Device**. Click **Device & Pin Options** and click the **Board Trace Model** tab (Figure 5–56).

Figure 5–56. Board Trace Model Tab of the Device and Pin Options Dialog Box





You can still click the **Capacitive Loading** tab. However, because you can configure all capacitive loading settings as part of the board trace model, the tab indicates that you must use the settings in the **Board Trace Model** tab.

All of the I/O standards available to the device are listed. Select any I/O standard from the list. The **Board trace model** list displays the names and values of all configurable components of the board trace for the selected

I/O standard. Components of the model are initially set to **short**, **open**, or a numeric value depending on the component. The default settings for components in the model for each I/O standard are device-specific and match the default test model used for calculating delay when the **Enable Advanced I/O Timing** option is turned off. In this way, default delay measurements are the same whether or not the **Enable Advanced I/O Timing** option is used.

For information about the default models used for measuring I/O delay, refer to the *DC & Switching Characteristics* chapter in the relevant device handbook.

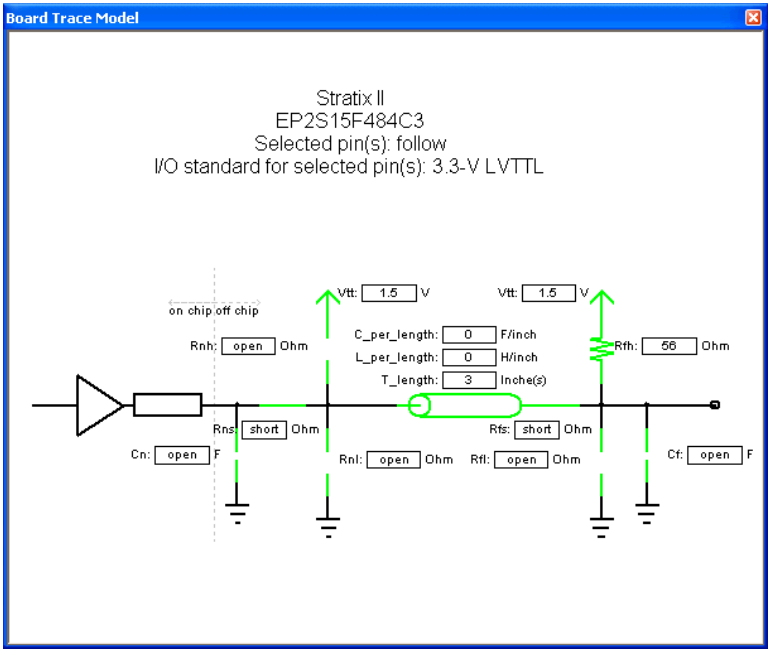
All of the component values listed in Figure 5–56 are adjustable. For differential I/O standards, the component values you set are used for both the positive and negative signals of a differential pair. An additional component, **Far differential resistance**, is also included. To reset individual settings to their defaults, leave the setting blank. If you want all the settings for an I/O standard to revert to their original settings, click **Reset**. Click **OK** to close the **Device & Pin Options** dialog box. Click **OK** again to close the **Settings** dialog box.

Any component value changes made in the **Board Trace Model** tab for a particular I/O standard are reflected in the Board Trace Model view in the Pin Planner of all pins assigned with the same I/O standard (described in "Customize the Board Trace Model in the Pin Planner"). However, custom component value changes made to selected pins in the Board Trace Model view in the Pin Planner take priority and are not affected by changes made to an I/O standard in the **Board Trace Model** tab.

Customize the Board Trace Model in the Pin Planner

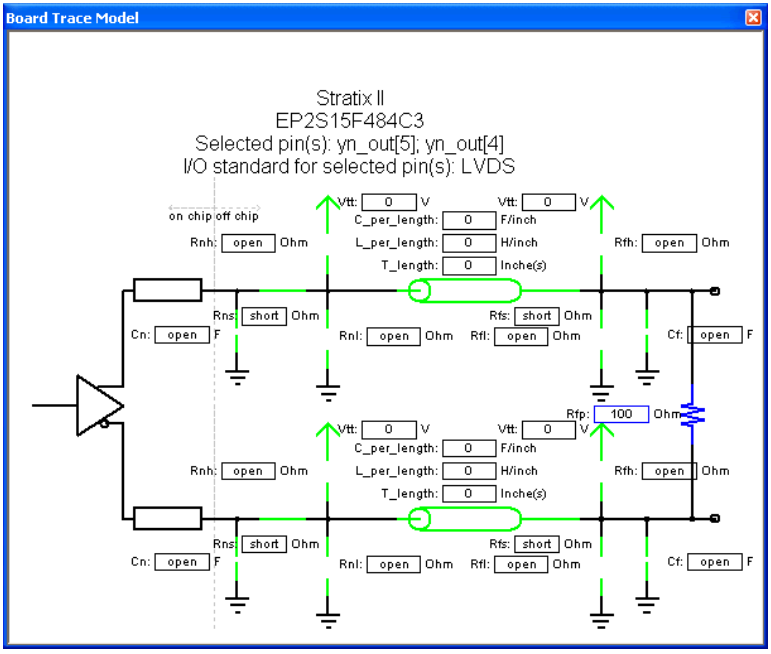
In addition to the views available in the Package View in the Pin Planner, you can also view a graphical representation of the board trace model you have configured using the Board Trace Model view. To open the Board Trace Model view, right-click on an output or bidirectional pin in the Groups List, the All Pins List, or the Package View and click **Board Trace Model**. The Board Trace Model view opens in a floating window (Figure 5–57).

Figure 5–57. Board Trace Model View



For differential signals, the Board Trace Model view displays the routing and components for both the positive and negative signals of the differential pair (Figure 5–58).

Figure 5–58. Differential Board Trace Model View



Any changes made to the Board Trace Model view for a differential signal pair must be performed on the positive signal of the pair. The settings must match between the positive and negative signals of a differential pair, so the changes are automatically reflected in the settings for the negative signal.

Double-click a component value to edit it. For numerical values, use standard unit prefixes such as *p*, *n*, and *k* to represent pico, nano, and kilo, respectively. To short across a series component or have an open circuit for a parallel component, double-click the component value and select **short** or **open** from the list.

For more details about configuring component values for a board trace model, including a complete list of the supported unit prefixes, refer to the Quartus II Help.

To view a display of the model for a particular pin, in the Package View, Groups List, or All Pins List, click on the pin. This changes the Board Trace Model view to display the model of the pin. To select multiple pins that share the same I/O standard, open the Board Trace Model view, and edit the model for all of the selected pins. If an input pin or multiple pins

with different I/O standards are selected, the Board Trace Model view window indicates that it cannot display the model for the selected pin or pins.

The components in the Board Trace Model view correspond to the components listed in the **Board Trace Model** tab directly, and the settings match initially. You can click and edit any value in the Board Trace Model view to customize the model for the selected pin or pins. Changes made in the Board Trace Model view do not affect the settings in the **Board Trace Model** tab.

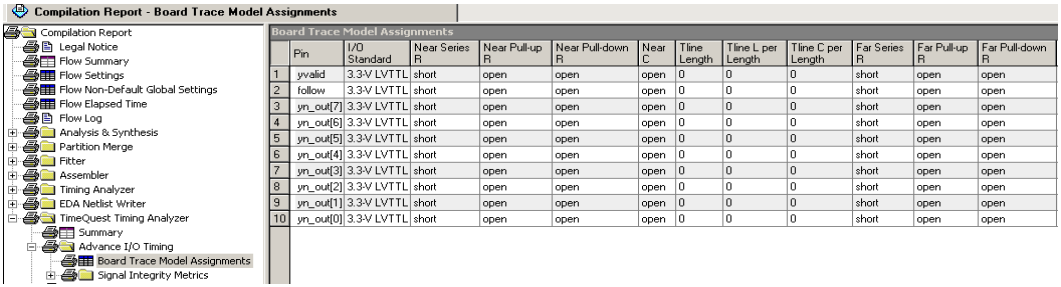
To configure board trace models for the pins in your design efficiently with these two methods of entry, define the model for each I/O standard in the **Board Trace Model** tab. With the overall model defined, use the Board Trace Model view in the Pin Planner to customize individual pins as needed. These customizations take priority over the settings in the **Board Trace Model** tab on a per pin and per model component basis, so they will not affect the settings on any other pin.

Create Signal Integrity Result Reports

After you have turned on **Enable Advanced I/O Timing** and configured board trace models for the pins you want to analyze, compile your project or run the Quartus II TimeQuest Timing Analyzer after a full compilation. The **Enable Advanced I/O Timing** option creates signal integrity sub-reports under TimeQuest Timing Analyzer in the Compilation Report window.

The Board Trace Model Assignments report (Figure 5–59) summarizes the board trace model component settings for each output and bidirectional signal.

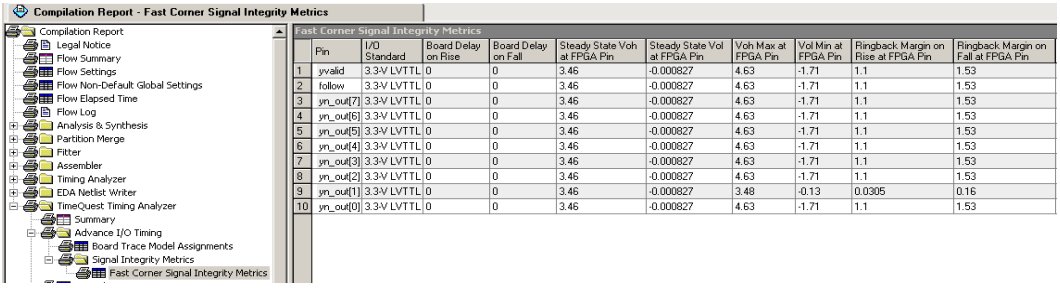
Figure 5–59. Board Trace Model Assignments Report



	Pin	I/O Standard	Near Series R	Near Pull-up R	Near Pull-down R	Near C	Tline Length	Tline L per Length	Tline C per Length	Far Series R	Far Pull-up R	Far Pull-down R
1	yvalid	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
2	follow	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
3	yn_out[7]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
4	yn_out[6]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
5	yn_out[5]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
6	yn_out[4]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
7	yn_out[3]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
8	yn_out[2]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
9	yn_out[1]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
10	yn_out[0]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open

The Signal Integrity Metrics subfolder contains detailed reports listing all of the metrics calculated by the **Advanced I/O Timing** option (Figure 5–60).

Figure 5–60. Example of Slow-Corner Signal Integrity Metrics Report



	Pin	I/O Standard	Board Delay on Rise	Board Delay on Fall	Steady State Voh at FPGA Pin	Steady State Vol at FPGA Pin	Voh Max at FPGA Pin	Vol Min at FPGA Pin	Ringback Margin on Rise at FPGA Pin	Ringback Margin on Fall at FPGA Pin
1	yvalid	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
2	follow	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
3	yn_out[7]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
4	yn_out[6]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
5	yn_out[5]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
6	yn_out[4]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
7	yn_out[3]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
8	yn_out[2]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
9	yn_out[1]	3.3-V LVTTL	0	0	3.46	-0.000827	3.48	-0.13	0.0305	0.16
10	yn_out[0]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53

The Slow- and Fast-Corner Signal Integrity Metrics reports are generated by the **Enable Advanced I/O Timing** option. They list, in tabular format, all of the signal integrity metrics calculated by the **Enable Advanced I/O Timing** option, based on the board trace model settings for each output or bidirectional pin. The reports contain many metrics, including measurements at both the FPGA and at the far-end load of board delay, steady state voltages, and rise and fall times.

The slow- or fast-corner reports are generated depending on the Timing Netlist option in the Quartus II TimeQuest Timing Analyzer. To select whether to create a slow- or a fast-corner report, in the TimeQuest Timing Analyzer on the Netlist menu, click **Create Timing Netlist**. Under Delay model, select **Slow corner** or **Fast corner** to create reports of that type.

For complete descriptions of all of the metrics calculated when the **Enable Advanced I/O Timing** option is turned on and diagrams illustrating the metrics on output waveforms, refer to the Quartus II Help. For more information about board level signal integrity and tips on how to improve signal integrity in your high-speed designs, refer to the Altera Signal Integrity Center.

For information about the configuration and use of the Quartus II TimeQuest Timing Analyzer, refer to the Quartus II Help or the *Timing Analysis* section in volume 3 of the *Quartus II Handbook*.

Conclusion

The Quartus II software provides many tools and features to help you with the I/O planning process. The I/O assignment analysis process offers the ability to validate pin assignments in all design stages, even before the development of the design. The ability to import and export assignments between the Quartus II software and other PCB tools also enables you to make iterative changes efficiently. Finally, the ability to enter a board trace model and create advanced timing reports based on how I/O signals are routed on a board truly makes the Quartus II software "board-aware."

Referenced Documents

The following documents were referenced in this chapter:

- *AN90: SameFrame Pin-Out Design for FineLine BGA Packages*
- *AN 315: Guidelines for Designing High-Speed FPGA PCBs*
- *altdq & altdqs Megafunction User Guide*
- *Altera Device Package Information Datasheet*
- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*
- *Cadence PCB Design Tools Support* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *DC & Switching Characteristics* chapter in volume 1 of the *Stratix II Device Handbook*
- *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*
- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Mentor Graphics PCB Design Tools Support* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Support for HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of *Quartus II Handbook*
- *Signal Integrity Analysis with Third Party Tools* chapter in volume 3 of the *Quartus II Handbook*
- *Stratix III Device Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Timing Analysis* section in volume 3 of the *Quartus II Handbook*

Document Revision History

Table 5–7 shows the revision history for this chapter.

Table 5–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Updated for the Quartus II software v7.2, including: • Added "Using the Live I/O Check Feature to Validate Pin Assignments" on page 5–65 about the new live I/O check feature.	Updated documentation to reflect the new live I/O check feature in the Quartus II 7.2 software.
May 2007 v7.1.0	• Updated I/O Planning Overview section. • Updated Tcl Scripts on page 5–61. • Updated Chip Planner or Timing Closure Floorplan on page 5–62. • Updated Using the Pin Planner on page 5–22. • Added Pin Migration View on page 5–34. • Updated Show I/O Banks on page 5–47. • Updated Using I/O Assignment Analysis to Validate Pin Assignments on page 5–67. • Added Configure User Nodes for Creating a Top-Level Design File on page 5–18. • Updated Document Revision History on page 5–99.	Updated documentation to reflect updates for the Quartus II 7.1 software. Updated figures.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	• Updated text and graphics to reflect GUI changes. • Added information about setting up and creating a top-level design file from megafunctions and IP MegaCores created in the Pin Planner. • Added spreadsheet functionality information to lists in the Pin Planner. • Added descriptions of new reports generated by I/O Assignment Analysis. • Added information the Advanced I/O Timing option, including the configuration of board trace models.	—
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0. • Updated text and graphics to reflect the GUI changes. • Added pin filtering information. • Added pin assignments and Pad View window information. • Added Package View information.	—
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1.0. • I/O Assignment Analysis material incorporated into chapter.	—
May 2005 v5.0.0	Initial release.	—

6. Mentor Graphics PCB Design Tools Support

QII52015-7.2.0

Introduction

With today's large, high-pin-count and high-speed FPGA devices, good and correct printed circuit board (PCB) design practices are more essential than ever for ensuring correct system operation. Typically, the PCB design takes place concurrently with the design and programming of the FPGA. Signal and pin assignments are initially made by the FPGA or ASIC designer, and the board designer must correctly transfer these assignments to the symbols used in their system circuit schematics and board layout. As the board design progresses, pin reassignments may be needed to optimize the PCB layout. These reassignments must in turn be relayed back to the FPGA designer so that the new assignments can be processed through an updated placement and routing of the FPGA design.

Mentor Graphics® provides tools to support this type of design flow. This chapter discusses how the Quartus® II software interacts with the Mentor Graphics I/O Designer software and the DxDesigner software to provide a completely cyclical FPGA-to-board integration design workflow. This chapter covers the following topics:

- General design flow between the Quartus II software, the Mentor Graphics I/O Designer software, and the DxDesigner software
- Setting up the Quartus II software to create the design flow files
- Creating an I/O Designer database project to incorporate the Quartus II software signal and pin assignment data
- Updating signal and pin assignment changes between the I/O Designer software and the Quartus II software
- Generating symbols in the I/O Designer software
- Creating symbols in the DxDesigner software from the Quartus II software output files without the use of the I/O Designer software

This chapter is intended primarily for board design and layout engineers who want to start the FPGA board integration while the FPGA is still in the design phase. Optionally, the board designer can plan the FPGA pinout and routing requirements in the Mentor Graphics tools and pass the information back to the Quartus II software for place-and-route. In addition, part librarians benefit from learning how to take output from the Quartus II software and use it to create new library parts and symbols.

The procedures in this chapter require the following software:

- The Quartus II software version 5.1 or higher
- DxDesigner software version 2004 or higher

Mentor Graphics I/O Designer software is optional.

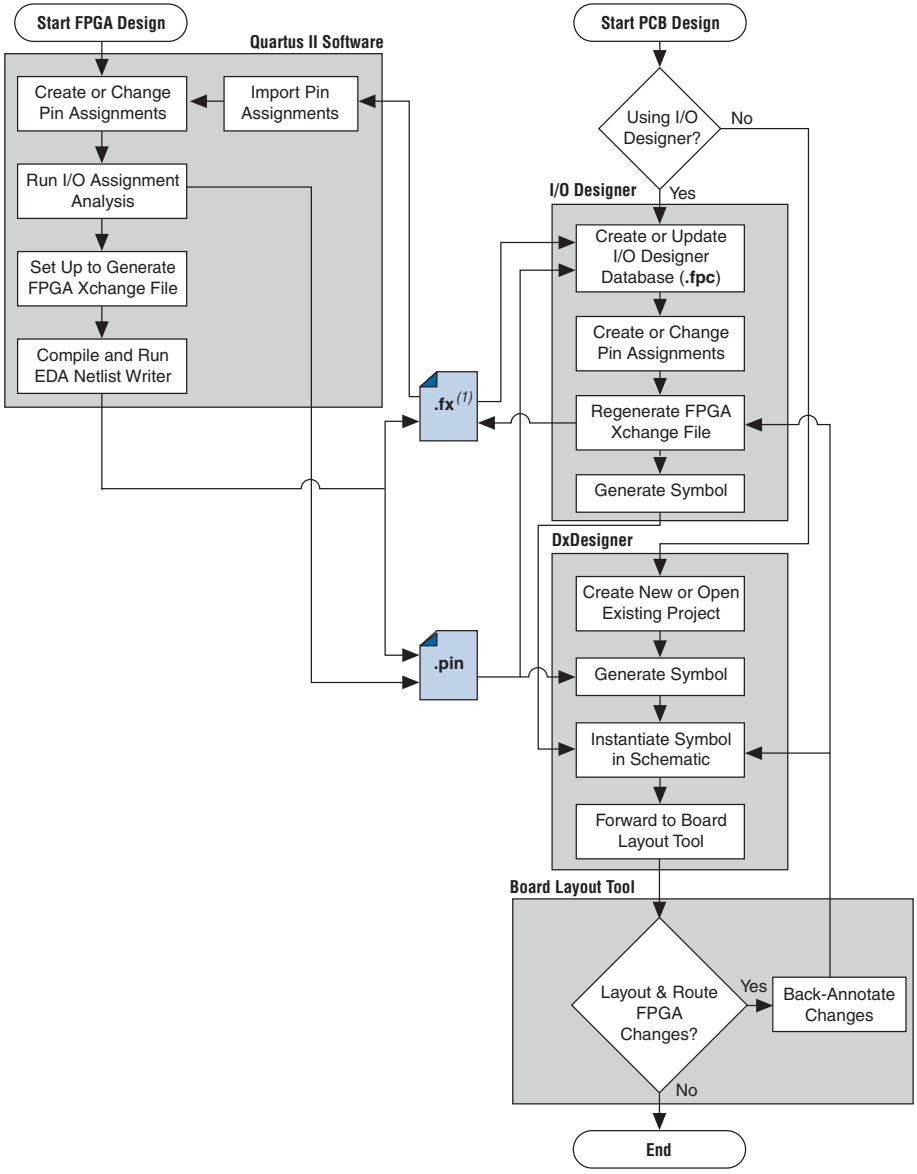


To obtain and license the Mentor Graphics tools and obtain product information, support, and training, go to the Mentor Graphics website at **www.mentor.com**.

FPGA-to-PCB Design Flow

In the examples in this section, you create a design flow integrating an Altera® FPGA design from the Quartus II software, and a circuit schematic in the DxDesigner software. Figure 6–1 shows the design flow with and without the I/O Designer software.

Figure 6–1. Design Flow with and without the I/O Designer Software



Note to Figure 6–1:

(1) The Quartus II software generates the FPGA Xchange file in the output directory you specify in the Board-Level Assignment Settings. However, the Quartus II software and the I/O Designer software can import pin assignments from an FPGA Xchange file located in any directory. Altera recommends that you work with a backup of the FPGA Xchange file to prevent overwriting existing assignments or importing invalid assignments.

The following tasks, which are described in this chapter, describe how to proceed through the design flow shown in Figure 6–1:

- Set up the board-level assignment settings to generate an FPGA Xchange file (**.fx**) for symbol generation in the Quartus II software
- Compile the design and generate the FPGA Xchange file and the Pin-Out file (**.pin**), which are located in the Quartus II project directory
- Create a board design using the DxDesigner software together with the I/O Designer software, which involves the following steps:
  - Create a new I/O Designer database based on the FPGA Xchange file and the Pin-Out file
  - Make adjustments to signal and pin assignments in the I/O Designer software
  - Regenerate the FPGA Xchange file in the I/O Designer software to reflect the I/O Designer software changes in the Quartus II software
  - Generate a single or fractured symbol for use in the DxDesigner software
  - Add the symbol to the **sym** directory of a DxDesigner project, or specify a new DxDesigner project with the new symbol
  - Instantiate the symbol in your DxDesigner schematic and export the design to the board layout tool
  - Back-annotate pin changes created in the board layout tool to the DxDesigner software and back to the I/O Designer software and the Quartus II software
- Create a board design using the DxDesigner software without the I/O Designer software, which involves the following steps:
  - Create a new DxBoardLink symbol using the Symbol Wizard and reference the Pin-Out file output from the Quartus II software in an existing DxDesigner project
  - Instantiate the symbol in your DxDesigner schematic and pass the design to a board layout tool

The I/O Designer software allows you to take advantage of the full FPGA symbol design, creation, editing, and back-annotation flow supported by Mentor Graphics tools.

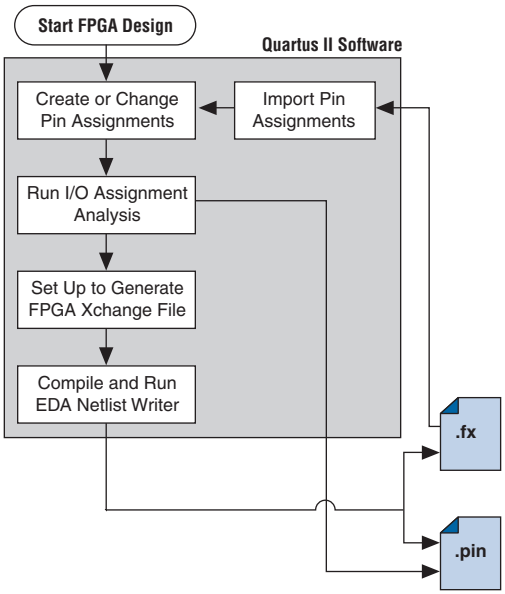


Symbols can be updated with design changes at any point with or without the I/O Designer software. However, if symbols are changed in the DxDesigner software, the I/O Designer software does not see the changes. If you change symbols using the DxDesigner software, you must reimport the symbols into I/O Designer to avoid overwriting your symbol changes.

Setting Up the Quartus II Software

You can transfer pin and signal assignments from the Quartus II software to the Mentor Graphics tools by generating two output files, a Pin-Out file (**.pin**) and an FPGA Xchange file (**.fx**) (Figure 6–2).

Figure 6–2. Pin-Out Files and FPGA Xchange Files *Note (1)*



Note to Figure 6–2:

(1) Refer to Figure 6–1 for the full design flow, which includes the I/O Designer software, the DxDesigner software, and the board layout tool flowchart details.

The two output files, the Pin-Out file and the FPGA Xchange file, are described in Table 6–1.

Table 6–1. Pin Assignment Output File Format Comparison

File Format	Description
Pin-Out file (**.pin**) *(1)*	An output file generated by the Quartus II Fitter. The file cannot be imported into the Quartus II software to change pin assignments. The file contains a complete list of the device pins including any unused I/O pins, and provides the following basic information fields for each assigned pin on a device: • Pin signal name/usage • Pin number • Signal direction • I/O standard • Voltage • I/O Bank • User or Fitter assigned
FPGA Xchange file (**.fx**) *(1)*,*(2)*	An input/output file generated by the Quartus II software and the I/O Designer software that can be imported and exported from both programs. Industry standard with room for future changes and additions. The FPGA Xchange file generated by the Quartus II software lists only assigned pins. The file provides the following advanced information fields for each pin on a device: • Pin number • I/O Bank • Signal name • Signal direction • I/O standard • Drive strength (mA) • Termination enabling • Slew rate • IOB Delay • Swap group • Differential pair type
	When generated by the I/O Designer software, all pins, including unused pins, are listed and the following fields are added: • Swap group • Differential pair type • Device pin name • Pin set • Pin set position • Pin set group • Super pin set group • Super pin set position

Notes to Table 6–1:

(1) For additional information about these file formats, refer to the Quartus II Help.

(2) For additional information about the information fields added by the Mentor Graphics software, refer to the Mentor Graphics website at www.mentor.com.

The I/O Designer software can also read from or update a Quartus II Settings File (**.qsf**). The Quartus II Settings File is used in the design flow in a similar manner to the FPGA Xchange file, but does not transfer pin swap group information between the I/O Designer software and the Quartus II software.



The **Quartus II Settings File** also contains additional important information about your project that is not used by the I/O Designer software. Because of this, Altera recommends that you use the FPGA Xchange file instead of the Quartus II Settings File for this design flow



For more information about the Quartus II Settings File, refer to the *Quartus II Settings File Reference Manual*.

Generating Pin-Out Files

The Quartus II Fitter generates the Pin-Out file whenever you perform a full compilation or I/O Assignment Analysis on your design. The file is generated and placed in your design directory and your file is named *<project name>***.pin**. The Mentor Graphics tools do not alter this file. The Quartus II software cannot import assignments from an existing Pin-Out file.

Generating FPGA Xchange Files

The FPGA Xchange file is not created automatically. To set up the Quartus II software to create the FPGA Xchange file, follow these steps:

1. Start the Quartus II software. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. Under EDA Tool Settings, click **Board-Level**. In the Board-Level Symbol Format list, choose **FPGA Xchange**.
3. Set the Output directory to the location where you want to save the file. The default output file path is *<project directory>***/symbols/fpgaxchange**. Click **OK**.
4. On the Processing menu, point to Start and click **Start EDA Netlist Writer**.

The output directory you selected is created when you generate the FPGA Xchange file.

CAUTION

Both the Quartus II software and the I/O Designer software can export and import an FPGA Xchange file. It is therefore possible to overwrite the FPGA Xchange file and import incorrect assignments into one or both programs. To prevent this occurrence from happening, make a backup copy of the file before importing, and import the copy instead of the file generated by the Quartus II software. In addition, assignments in the Quartus II software can be protected by following the steps in "Protecting Assignments in the Quartus II Software" on page 6–22.

Creating a Backup Quartus II Settings File

To create a backup Quartus II Settings File, perform the following steps:

1. On the Assignments menu, click **Import Assignments**. The **Import Assignments** dialog box appears.
2. In the **Import Assignments** dialog box, browse to your project and turn on **Copy existing assignments into** <*project name*>**.qsf.bak**.
3. Click **OK**.

Following these steps automatically creates a backup Quartus II Settings File of your current pin assignments.

For more information about pin and signal assignment transfer, and files the Quartus II software can import and export, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

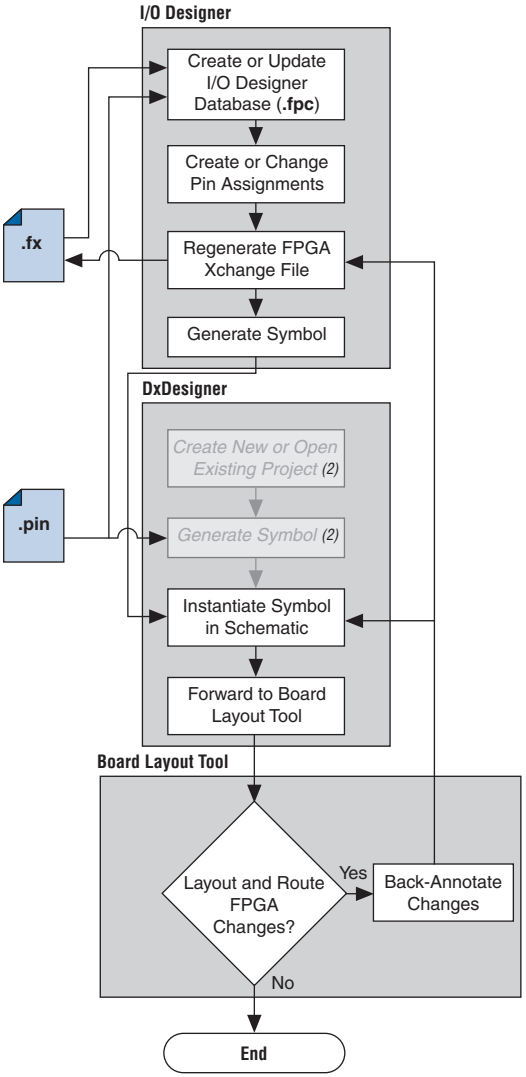
FPGA-to-Board Integration with the I/O Designer Software

The Mentor Graphics I/O Designer software allows you to integrate your FPGA and PCB designs. Pin and signal assignment changes can be made anywhere in the design flow, typically using either the Quartus II Pin Planner or the I/O Designer software. The I/O Designer software facilitates moving these changes, as well as synthesis, placement, and routing changes, between the Quartus II software, an external synthesis tool (if used), and a schematic capture tool such as the DxDesigner software.

This section describes how to use the I/O Designer software to transfer pin and signal assignment information to and from the Quartus II software with the FPGA Xchange file, and how to create symbols for the DxDesigner software.

Figure 6–3 shows the design flow using the I/O Designer software.

Figure 6–3. Design Flow Using the I/O Designer Software Note (1)



Notes to Figure 6–3:

(1) Refer to Figure 6–1 for the full design flow including the Quartus II software flowchart details.

(2) These are DxDesigner software-specific steps in the design flow and are not part of the I/O Designer flow.

For more information about the I/O Designer software, and to obtain usage, support, and product updates, use the Help menu in the I/O Designer software or refer to the Mentor Graphics website at **www.mentor.com**.

I/O Designer Database Wizard

All I/O Designer project information is stored in an I/O Designer Database (**.fpc**) file. You can create a new database that incorporates the FPGA Xchange file and Pin-Out file information generated by the Quartus II software by using the I/O Designer Database Wizard. You can also create a new, empty database and manually add the assignment information. If there is no signal or pin assignment information currently available, you can create an empty database that contains only a selection of the target device. This is useful if you know the signals in your design and the pins you want to assign. You can transfer this information at a later time to the Quartus II software for place-and-route.

It is possible to create an I/O Designer database with only one type of file or the other. However, if only a Pin-Out file is used, any I/O assignment changes made in the I/O Designer software cannot be imported back into the Quartus II software without first generating an FPGA Xchange file. If only an FPGA Xchange file is used to create the I/O Designer database, the database may not contain a complete picture of all of the I/O assignment information available. The FPGA Xchange file generated by the Quartus II software only lists pins with assigned signals. Since the Pin-Out file lists all device pins—whether signals are assigned to them or not—its use, along with the FPGA Xchange file, produces the most complete set of information for creating the I/O Designer Database.

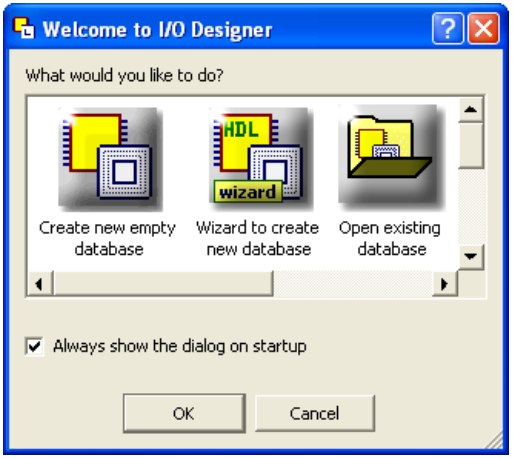
To create a new I/O Designer database using the Database Wizard, perform the following steps:

If you skip a step in this process, you can complete the skipped step later, filling in the appropriate information. To return to a skipped step, on the Properties menu, click **File**.

1. Start the I/O Designer software. The **Welcome to I/O Designer** dialog box appears (Figure 6–4). Select **Wizard to create new database** and click **OK**.

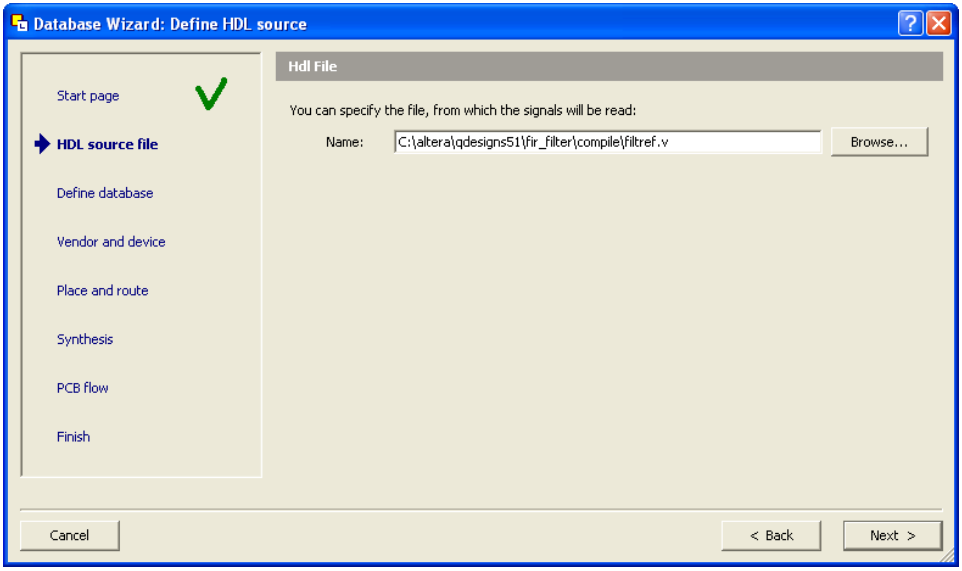
   If the **Welcome to I/O Designer** dialog box is not shown because it was disabled, you can access the Wizard through the menus. To access the Wizard on the File menu, click **Database Wizard**.

Figure 6–4. I/O Designer Welcome Dialog Box



2. Click **Next**. The **Define HDL source file** page opens (Figure 6–5).

Figure 6–5. Database Wizard HDL File Page



For more information about creating and using HDL files in the Quartus II software, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*, or refer to the I/O Designer Help.

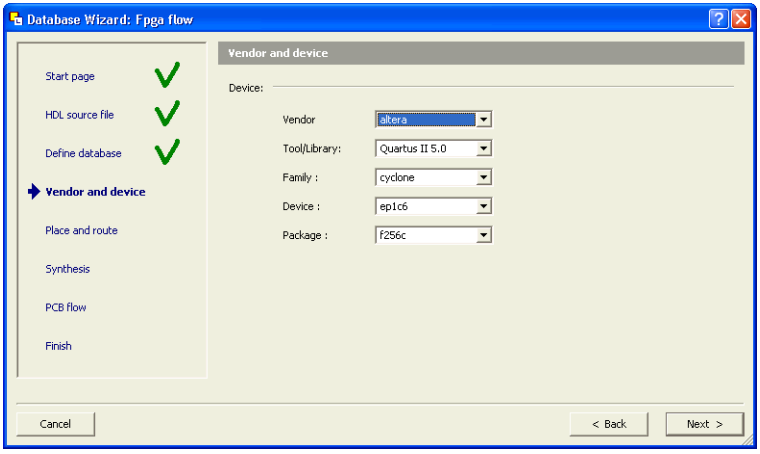
If no HDL files are available, or if your signal and pin assignments are already contained in the FPGA Xchange file, you do not have to complete step 3 and can proceed to step 4.

3. If you have created a Verilog HDL or VHDL file in your Quartus II software design, you can enter a top-level Verilog HDL or VHDL file. Adding a file allows you to create functional blocks or get signal names from your design. All physical pin assignments must be created in I/O Designer if no FPGA Xchange file or Pin-Out file is used. Click **Next**. The Database Name page is shown.

4. In the Database Name window, enter your database file name. Click **Next**. The Database Location window is shown.

5. Enter a path to the new database or an existing one in the **Location** field, or browse to a database location. Click **Next**. The **FPGA flow** page is shown (Figure 6–6).

Figure 6–6. Database Wizard Vendor and Device Page



6. In the Vendor menu, click **Altera**.

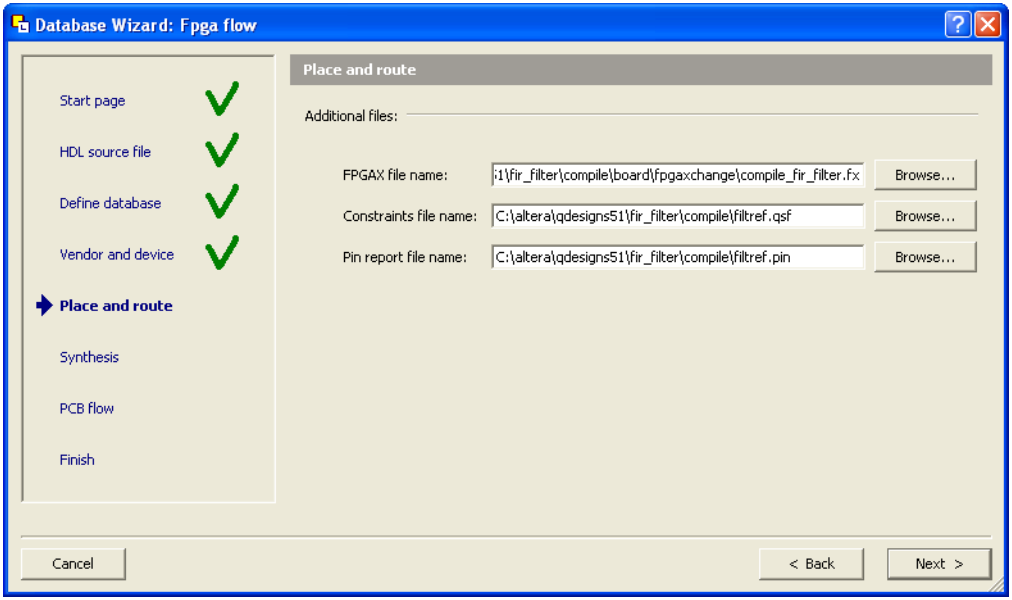
7. In the Tool/Library menu, click **Quartus II 5.0**, or a later version of the Quartus II software.

8. Select the appropriate device family, device, package, and speed (if applicable), from the corresponding menus. Click **Next**. The **Place and route** page is shown (Figure 6–7).



The Quartus II software version selections in the Tool/Library menu may not reflect the version of the Quartus II software currently installed on your system even if you are using the most current version of the I/O Designer software. The version number selection in this window is used in the I/O Designer software to identify the devices that were available or obsolete in that particular version of the Quartus II software. If you are unsure of the version to select, use the most recent version listed in the menu. If the device you are targeting does not appear in the device menu after making this selection, the device may be new and not yet added to the I/O Designer software. For I/O Designer software updates, contact Mentor Graphics or refer to their website at **www.mentor.com**.

Figure 6–7. Database Wizard Place and Route Page



9. In the **FPGAX file name** field, type or browse to the backup copy of the FPGA Xchange file generated by the Quartus II software.

10. In the **Pin report file name** field, type or browse to the Pin-Out file generated by the Quartus II software. Click **Next**.

    In addition, you can select a Quartus II Settings File for update. The I/O Designer software can update the pin assignment information in the Quartus II Settings File without affecting any other information contained in the file.

    You can select a Pin-Out file without selecting an FPGA Xchange file for import. The I/O Designer software does not generate a Pin-Out file. To transfer assignment information to the Quartus II software, select an additional file and file type. Altera recommends selecting an FPGA Xchange file in addition to a Pin-Out file for transferring all of the assignment information contained within both types of files.

    In some versions of the I/O Designer software, the standard file picker may incorrectly look for a Pin-Out file instead of an FPGA Xchange file. In this case, select **All Files (*.*)** from the **Save as type** list and select the file from the list.

11. The **Synthesis** page displays. On the **Synthesis** page, you can specify an external synthesis tool and a synthesis constraints file for use with the tool. If you do not use an external synthesis tool, click **Next**.

For more information about third-party synthesis tools, refer to volume 3 of the *Quartus II Handbook*.

12. The **PCB Flow** page is shown (Figure 6–8). On the **PCB Flow** page, you can select an existing schematic project or create a new project as a destination for symbol information.

    - To select an existing project, select **Choose existing project** and click **Browse** after the Project Path field. The **Select project** dialog box appears. Select the project.
    - To create a new project, in the **Select project** dialog box, select **Create new empty project**. Enter the project file name in the **Name** field and browse to the location where you want to save the file (Figure 6–9). Click **OK**.

Figure 6–8. PCB Flow Page

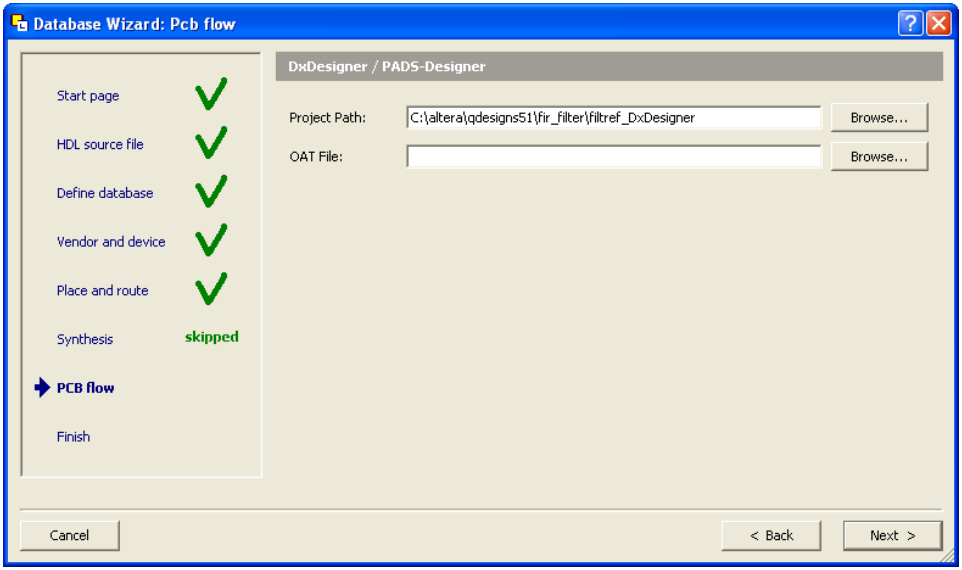
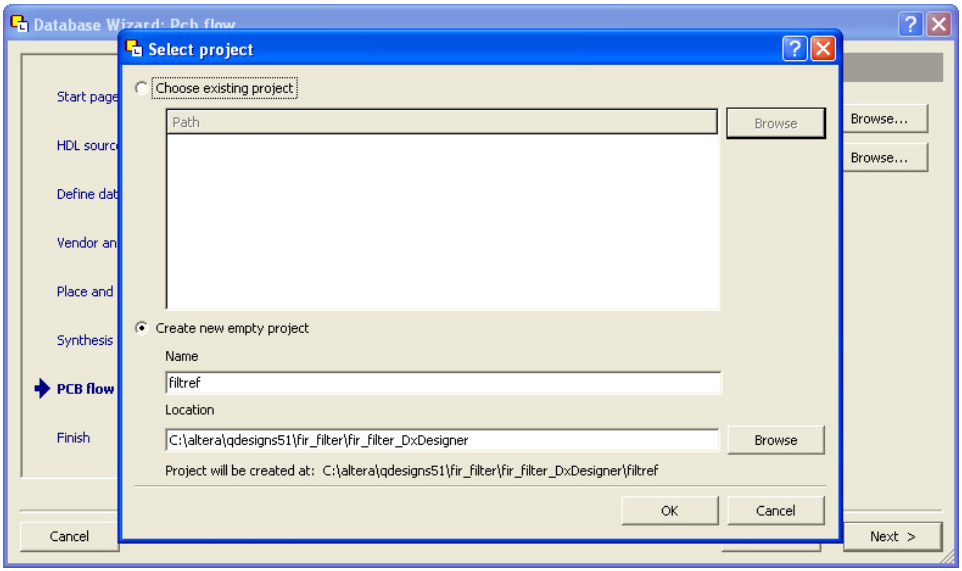


Figure 6–9. Select Project Dialog Box

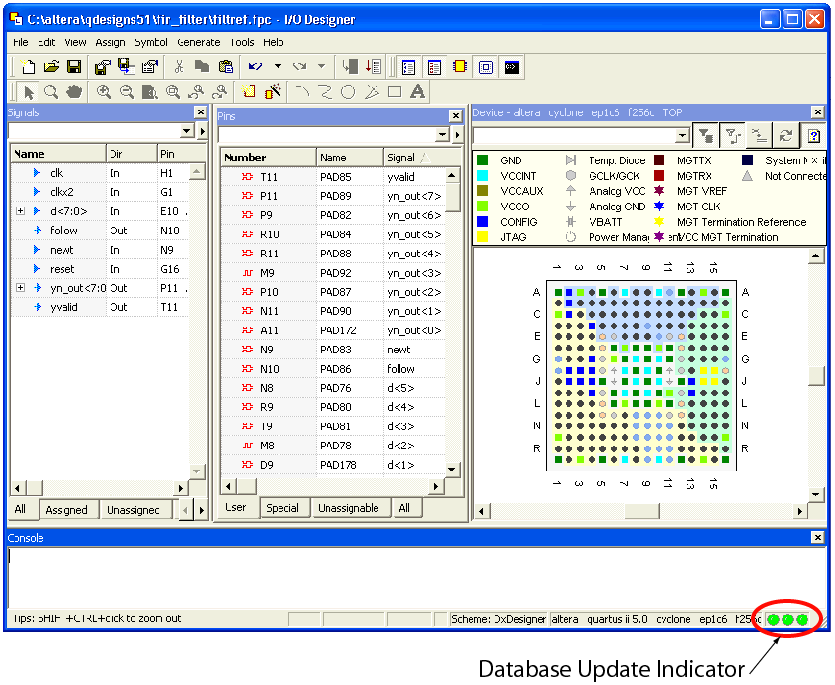


If you have not specified a design tool for sending symbol information to in I/O Designer, click **Advanced** in the **PCB Flow** page and select your design tool. If the DxDesigner software is selected, you have the option of specifying a Hierarchical Occurrence Attributes (**.oat**) file to import into the I/O Designer software (Figure 6–8). Click **Next**, then click **Finish** to create the database.

In I/O Designer version 2005 or later, the Update Wizard (refer to Figure 6–13 on page 6–20) is shown when you finish creating the database using the database wizard. Use the Update Wizard to confirm creation of the I/O Designer database using the selected FPGA Xchange and Pin-Out files.

Use the I/O Designer software and your newly created database to make pin assignment changes, create pin swap groups, or adjust signal and pin properties in the I/O Designer GUI (Figure 6–10).

Figure 6–10. I/O Designer Main Window

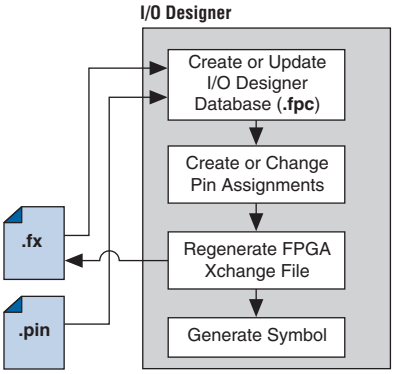


For more information about using the I/O Designer software and the DxDesigner software, refer to the Mentor Graphics website at **www.mentor.com** or refer to the I/O Designer software or the DxDesigner Help.

Updating Pin Assignments from the Quartus II Software

As the design process continues, the FPGA designer may need to make changes to the logic design in the Quartus II software that place signals on different pins after the design is recompiled, or manually by using the Quartus II Pin Planner. These types of changes must be carried forward to the circuit schematic and board layout tools to ensure that signals are connected to the correct pins on the FPGA. Updating the FPGA Xchange file and the Pin-Out file in the Quartus II software facilitates this flow (Figure 6–11).

Figure 6–11. Updating the I/O Designer Pin Assignments in the Design Flow *Note (1)*



Note to Figure 6–11:

(1) Refer to Figure 6–1 for the full design flow, which includes the Quartus II software, the DxDesigner software, and the board layout tool flowchart details.

To update the FPGA Xchange file and the Pin-Out file in the Quartus II software after making changes to the design, run a full compilation, or on the Start menu, point to Processing and click **Start EDA Netlist Writer**. The FPGA Xchange file in your selected output directory and the Pin-Out file in your project directory are updated. You must rerun the I/O Assignment Analyzer whenever you make I/O changes in the Quartus II software. To rerun the I/O Assignment Analyzer, on the Processing menu, click **Start Compilation**, or to run a full compilation, on the Processing menu, point to Start and click **Start I/O Assignment Analysis**.

Refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook* for more information about setting up the FPGA Xchange file and running the I/O Assignment Analyzer.

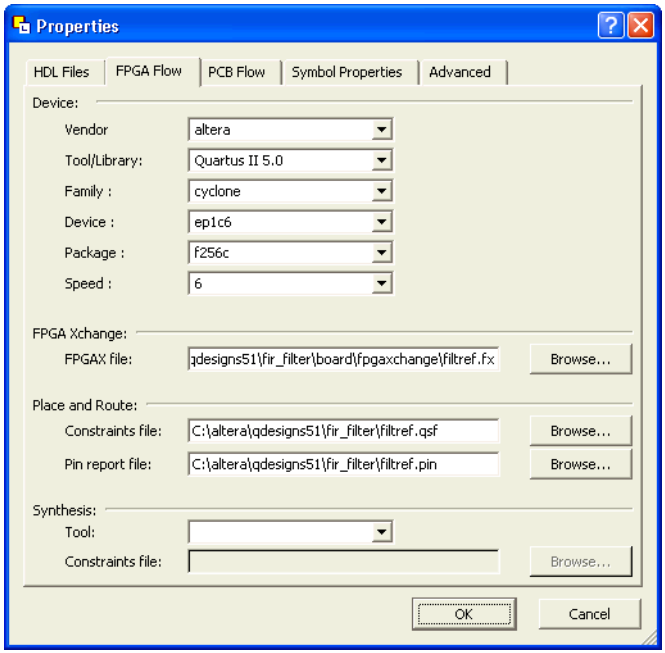
CAUTION

If your I/O Designer database points to the FPGA Xchange file generated by the Quartus II software instead of a backup copy of the file, updating the file in the Quartus II software overwrites any changes made to the file by the I/O Designer software. If there are I/O Designer assignments in the FPGA Xchange file that you want to preserve, create a backup copy of the file before updating it in the Quartus II software, and verify that your I/O Designer database points to the backup copy. To point to the backup copy, perform the steps in the following section.

Whenever the FPGA Xchange file or the Pin-Out file is updated in the Quartus II software, the changes can be imported into the I/O Designer database. You must set up the locations for the files in the I/O Designer software.

1. To set up the file locations if they are not already set, on the File menu, click **Properties**. The project **Properties** dialog box appears (Figure 6–12).

Figure 6–12. Project Properties Dialog Box



2. Under **FPGA Xchange**, click **Browse** to select the FPGA Xchange file name and file location.

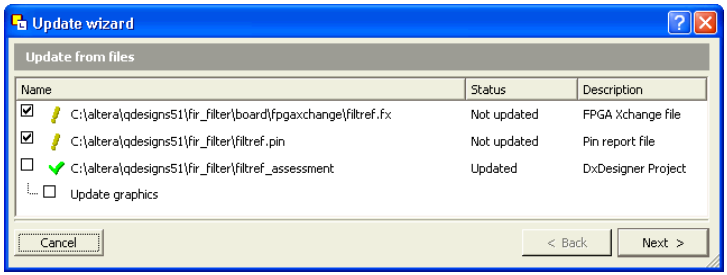
3. To specify a Pin report file, under **Place and Route**, click **Browse** to select the Pin-Out file name and file location.

Once you have set up these file locations, the I/O Designer software monitors these files for changes. If the FPGA Xchange file or Pin-Out file changes during the design flow, three indicators flash red in the lower right-hand corner of the I/O Designer main window (see Figure 6–10 on page 6–16). You can continue working or click on the indicators to open the I/O Designer Update Wizard. If you have made changes to your design in the Quartus II software that result in an updated FPGA Xchange file or Pin-Out file and the update indicators do not flash or you have previously canceled an indicated update, manually open the Update Wizard. To open the Wizard, on the File menu, click **Update**.

In versions of the I/O Designer software before version 2005, instead of using flashing indicators, the I/O Designer software displays a dialog box asking if you want to open the Update Wizard.

The I/O Designer Update Wizard lists the updated files associated with the database (Figure 6–13).

Figure 6–13. Update Wizard Dialog Box



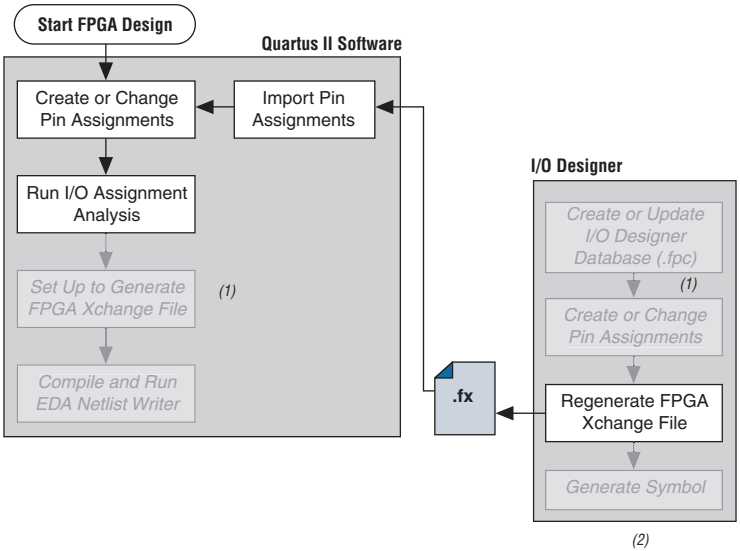
The paths to the updated files have yellow exclamation points and the **Status** column shows **Not updated**, indicating that the database has not yet been updated with the newer information contained in the files. A checkmark to the left of any updated file indicates that the file will update the database. Turn on any files you want to use to update the I/O Designer database, and click **Next**. If you are not satisfied with the database update, on the Edit menu, click **Undo**.

You can update the I/O Designer database using both the FPGA Xchange file and the Pin-Out file at the same time. Turning on both the FPGA Xchange file and the Pin-Out file for update causes the Update Wizard to provide options for using assignments from one file or the other exclusively or merging the assignments contained in both files into the I/O Designer database. Versions of the I/O Designer software older than version 2005 simply merge assignments contained in multiple files.

Sending Pin Assignment Changes to the Quartus II Software

In the same way that the FPGA designer can make adjustments that affect the PCB design, the board designer can make changes to optimize signal routing and layout that must be applied to the FPGA. The FPGA designer can take these required changes back into the Quartus II software to refit the logic to match the adjustments to the pinout. The I/O Designer software can accommodate this reverse flow as shown in Figure 6–14.

Figure 6–14. Updating the Quartus II Pin Assignments in the Reverse Design Flow



Notes to Figure 6–14:

(1) These are software-specific steps in the design flow and are not necessary for the reverse flow steps of the design.

(2) Refer to Figure 6–1 for the full design flow, which includes the complete I/O Designer software, the DxDesigner software, and the board layout tool flowchart details.

Pin assignment changes are made directly in the I/O Designer software, or the software automatically updates changes made in a board layout tool that are back-annotated to a schematic entry program such as the DxDesigner software. You must update the FPGA Xchange file to reflect these updates in the Quartus II software. To perform this update in the I/O Designer software, on the Generate menu, click **FPGA Xchange File**.

CAUTION

If your I/O Designer database points to the FPGA Xchange file generated by the Quartus II software instead of a backup copy, updating the file from the I/O Designer software overwrites any changes that may have been made to the file by the Quartus II software. If there are assignments from the Quartus II software in the file that you want to preserve, make a backup copy of the file before updating it in the I/O Designer software, and verify that your I/O Designer database points to the backup copy. To point to the backup copy, perform the steps in "Updating Pin Assignments from the Quartus II Software" on page 6–17.

After the FPGA Xchange file is updated, you must import it into the Quartus II software. To import the file, perform the following steps:

1. Start the Quartus II software and open your project.
2. On the Assignments menu, click **Import Assignments**.
3. In the File name box, click **Browse** and from the **Files of type** list, select **FPGA Xchange Files (*.fx)**.
3. Select the FPGA Xchange file and click **Open**.
4. Click **OK**.

CAUTION

Both the Quartus II software and the I/O Designer software can export and import an FPGA Xchange file. It is therefore possible to overwrite the FPGA Xchange file and import incorrect assignments into one or both programs. To prevent this occurrence from happening, make a backup copy of the file before importing, and import the copy instead of the file generated by the Quartus II software. In addition, assignments in the Quartus II software can be protected by following the steps in "Protecting Assignments in the Quartus II Software".

Protecting Assignments in the Quartus II Software

To protect assignments in the Quartus II software, use the following steps:

1. Start the Quartus II software.
2. On the Assignments menu, click **Import Assignments**. The **Import Assignments** dialog box appears.
3. Turn on **Copy existing assignments into** <*project name*>**.qsf.bak before importing** before importing the FPGA Xchange file. This action automatically creates a backup Quartus II constraints file that contain all of your current pin assignments.

Generating Symbols for the DxDesigner Software

Along with circuit simulation, circuit board schematic creation is one of the first tasks required in the design of a new PCB. Schematics are required to understand how the PCB will work, and to generate a netlist that is passed to a board layout tool for board design and routing. The I/O Designer software provides the ability to create schematic symbols based on the FPGA design exported from the Quartus II software.

Most FPGA devices contain hundreds of pins, requiring large schematic symbols that may not fit on a single schematic page. Symbol designs in the I/O Designer software can be split or fractured into a number of functional blocks, allowing multiple part fractures on the same schematic page or across multiple pages. In the DxDesigner software, these part fractures are joined together with the use of the `HETERO` attribute.

The I/O Designer software can generate symbols for use in a number of Mentor Graphics schematic entry tools, and can import changes back-annotated by board layout tools to update the database and feed updates back to the Quartus II software using the FPGA Xchange file. This section discusses symbol creation specifically for the DxDesigner software.

Schematic symbols are created in the I/O Designer software in the following ways:

- Manually
- Using the I/O Symbol Wizard
- Importing previously created symbols from the DxDesigner software

The I/O Designer Symbol Wizard can be used as a design base that allows you to quickly create a symbol for manual editing at a later time. If you have already created symbols in a DxDesigner project and want to apply a different FPGA design to them, you can manually import these symbols from the DxDesigner project. To import the symbols, open the I/O Designer software, and on the File menu, click **Import Symbol**.



For more information about importing symbols from the DxDesigner software into an I/O Designer database, refer to the I/O Designer Help.

Symbols created in the I/O Designer software are either functional, physical (PCB), or a combination of functional and physical. A functional symbol is based on signals imported into the database, usually from Verilog HDL or VHDL files. No physical device pins must be associated with the signals to generate a functional symbol. This section focuses on board-level PCB symbols with signals directly mapped to physical device pins through assignments in either the Quartus II Pin Planner or in the I/O Designer database.



For information about manually creating symbols, importing symbols, and editing symbols in the I/O Designer software, as well as the different types of symbols the software can generate, refer to the I/O Designer Help.

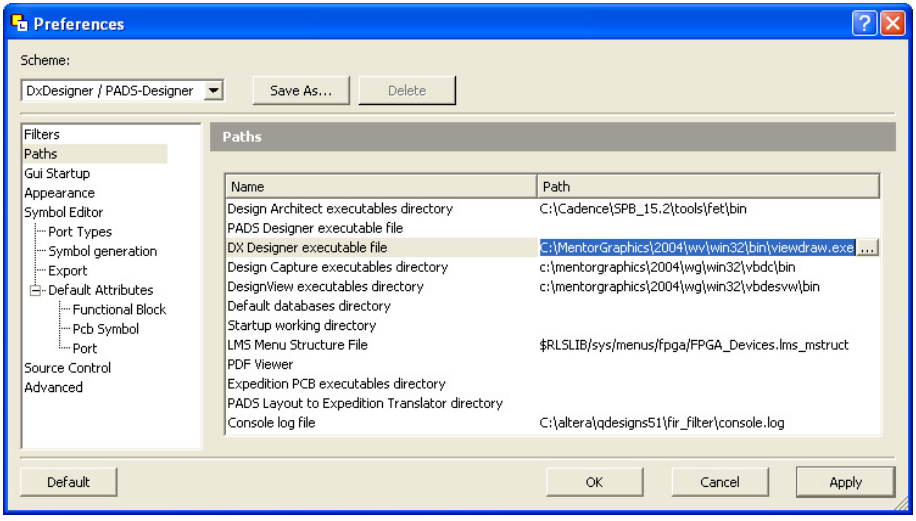
Setting Up the I/O Designer Software to Work with the DxDesigner Software

If you created your I/O Designer database using the Database Wizard, you may already be set up to export symbols to a DxDesigner project. To verify this, or to manually set up the I/O Designer software to work with the DxDesigner software, you must set the path to the DxDesigner executable, set the export type to DxDesigner, and set the path to a DxDesigner project directory.

To set these options, perform the following steps:

1. Start the I/O Designer software.
2. On the Tools menu, click **Preferences**. The **Preferences** dialog box appears.
3. Click **Paths**, double-click on the DxDesigner executable file path field, and click **Browse** to select the location of the DxDesigner application (Figure 6–15).
4. Click **Apply**.

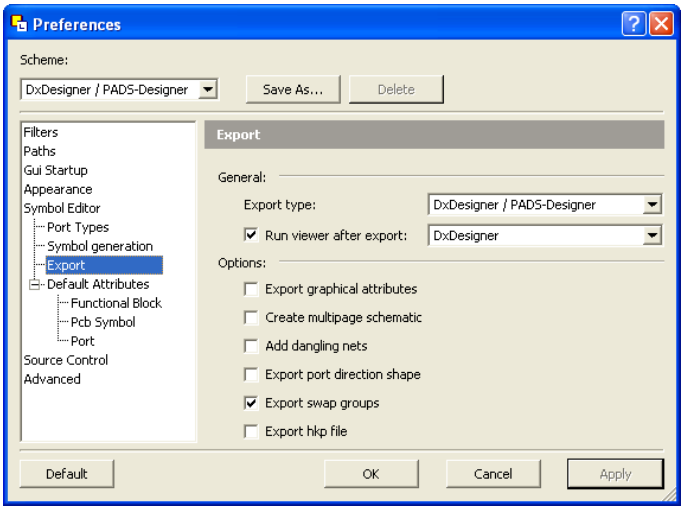
Figure 6–15. Path Preferences Dialog Box



5. Click **Symbol Editor** and click **Export**. In the Export type menu, under General, select **DxDesigner/PADS-Designer** (Figure 6–16).

6. Click **Apply** and click **OK**.

Figure 6–16. Symbol Editor Export Preferences



7. On the File menu, click **Properties**. The project **Properties** dialog box appears.

8. Click the **PCB Flow** tab and click **Path to a DxDesigner project directory**.

9. Click **OK**.

If you did not create a new DxDesigner project in the Database Wizard and you do not already have a DxDesigner project, you must create a new database using the DxDesigner software, and point the I/O Designer software to this new project.

For information about creating and working with DxDesigner projects, refer to the DxDesigner Help.

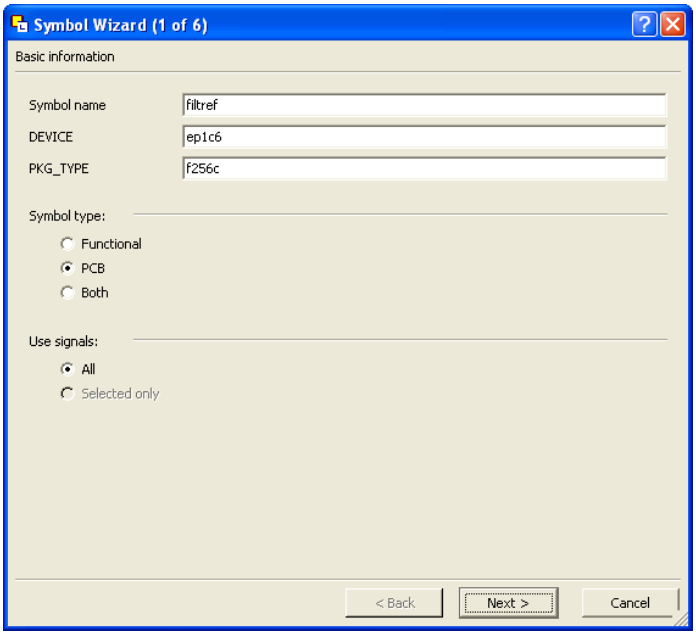
Create Symbols with the Symbol Wizard

FPGA symbols based on Altera devices can be created, fractured, and edited using the I/O Designer Symbol Wizard. To create a symbol based on a selected Altera FPGA device:

1. Start the I/O Designer software.

2. Click **Symbol Wizard** in the toolbar, or on the Symbol menu, click **Symbol Wizard**. The **Symbol Wizard (1 of 6) page** is shown (Figure 6–17).

Figure 6–17. Symbol Wizard



3. On the first Symbol Wizard page, in the **Symbol name** field, enter the symbol name. The **DEVICE** and **PKG_TYPE** fields are populated with the device and package information automatically. Under Symbol type, click **PCB**. Under Use signals, click **All**.

4. Click **Next**. The **Symbol Wizard (2 of 6)** page is shown.



If the **DEVICE** and **PKG_TYPE** fields are blank or incorrect, cancel the Symbol Wizard and select the correct device information. On the File menu, click **Properties**. In the Properties window, click the **FPGA Flow** tab and enter the correct device information.

5. On page 2 of the Symbol Wizard, select fracturing options for your symbol. If you are using the Symbol Wizard to edit a previously created fractured symbol, you must turn on **Reuse existing fractures** so that your current fractures are not altered. Select other options on this page as appropriate for your symbol.

6. Click **Next**. The **Symbol Wizard (3 of 6)** page is shown.

7. Additional fracturing options are available on page 3 of the Symbol Wizard. After selecting the desired options, click **Next**. The **Symbol Wizard (4 of 6)** page is shown.

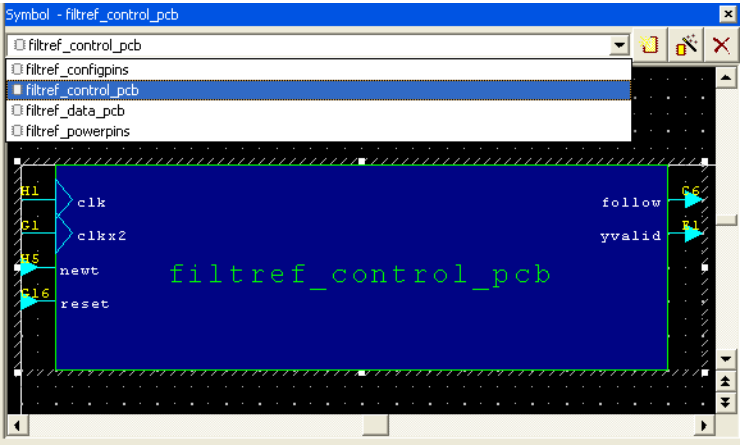
8. On page 4 of the Symbol Wizard, select the options for how the symbols will look. Select the desired options and click **Next**. The **Symbol Wizard (5 of 6)** page is shown.

9. On page 5 of the Symbol Wizard, define what information will be labeled for the entire symbol and for individual pins. Select the desired options and click **Next**. The **Symbol Wizard (6 of 6)** page is shown.

10. On the final page of the Symbol Wizard, add additional signals and pins that have not already been placed in the symbol. Click **Finish** when you complete your selections.

Your symbol is complete. You can view your symbol and any fractures you created using the Symbol Editor (Figure 6–18). You can edit parts of the symbol, delete fractures, or rerun the Symbol Wizard.

Figure 6–18. The I/O Designer Symbol Editor



If assignments in the I/O Designer database are updated, the symbols created in the I/O Designer software automatically reflect these changes. Assignment changes can be made within the I/O Designer software, with an updated FPGA Xchange file from the Quartus II software, or from a back-annotated change in your board layout tool.

Export Symbols to the DxDesigner Software

After you have completed your symbols, export the symbols to your DxDesigner project. To generate all the fractures of a symbol, on the Generate menu, click **All Symbols**. To generate a symbol for the currently displayed symbol in Symbol Editor, click **Current Symbol Only**. Each symbol in the database is saved as a separate file in the **/sym** directory in your DxDesigner project. The symbols can be instantiated in your DxDesigner schematics.

For more information about working with DxDesigner projects, refer to the DxDesigner Help.

Scripting Support

The I/O Designer software features a command line Tcl interpreter. All commands issued through the GUI in the I/O Designer software are translated into Tcl commands that are run by the tool. You can view the generated Tcl commands and run scripts, or enter individual commands in the I/O Designer Console window.

The following section includes commands that perform some of the operations described in this chapter.

If you want to change the FPGA Xchange file from which the I/O Designer software updates assignments, type the following command at an I/O Designer Tcl prompt:

```
set_fpga_xchange_file <file name>
```

After the FPGA Xchange file is specified, use the following command to update the I/O Designer database with assignment updates made in the Quartus II software:

```
update_from_fpga_xchange_file
```

Use the following command to update the FPGA Xchange file with changes made to the assignments in the I/O Designer software for transfer back into the Quartus II software:

```
generate_fpga_xchange_file
```

If you want to import assignment data from a Pin-Out file created by the Quartus II software, use the following command:

```
set_pin_report_file -quartus_pin <file name>
```

Run the I/O Designer Symbol Wizard with the following command:

```
symbolwizard
```

Set the DxDesigner project directory path where symbols are saved with the following command:

```
set_dx_designer_project -path <path>
```



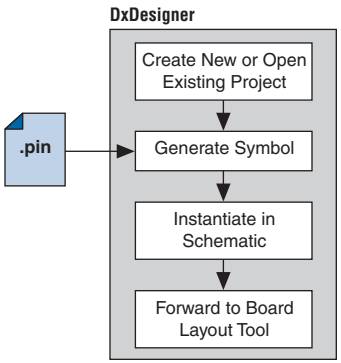
For more information about Tcl scripting and Tcl scripting with the Quartus II software, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about the Tcl scripting capabilities of the I/O Designer software as well as a list of all the commands available, refer to the I/O Designer Help.

FPGA-to-Board Integration with the DxDesigner Software

The Mentor Graphics DxDesigner software is a design entry tool for schematic capture. You can use it to create flat circuit schematics for all types of PCB design. You can also use the DxDesigner software to create hierarchical schematics that facilitate design reuse and a team-based design. You can use the DxDesigner software in the design flow alone or in conjunction with the I/O Designer software. However, if you use the DxDesigner software without the I/O Designer software, the design flow is one-way, using only the Pin-Out file generated by the Quartus II software.

Signal and pin assignment changes can be made only in the Quartus II software and are reflected in updated symbols in a DxDesigner schematic. You cannot back-annotate changes made in a board layout tool or in a DxDesigner symbol to the Quartus II software. Figure 6–19 shows the design flow when the I/O Designer software is not used.

Figure 6–19. Design Flow Without the I/O Designer Software *Note (1)*



Note to Figure 6–19:

(1) Refer to Figure 6–1 for the full design flow, which includes the Quartus II software, the I/O Designer software, and the board layout tool flowchart details.

For more information about the DxDesigner software, including usage, support, training, and product updates, refer to the Mentor Graphics web page at **www.mentor.com**, or choose Schematic Design Help Topics in the DxDesigner Help.

DxDesigner Project Settings

New projects in the DxDesigner software are already set up to create FPGA symbols by default. However, for complete support and compatibility with the I/O Designer software, if it is used with the DxDesigner software, you should enable the DxBoardLink Flow options.

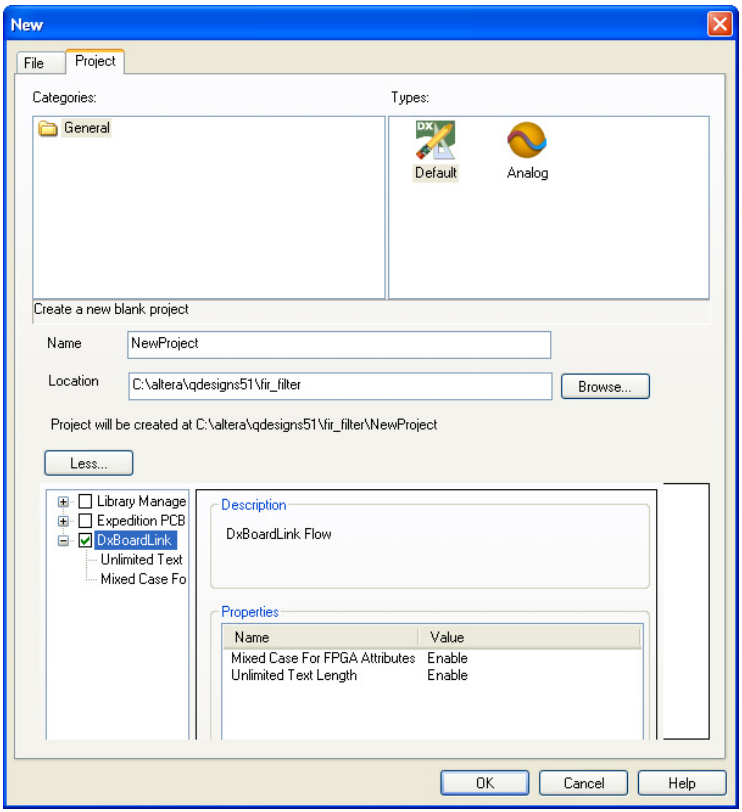
You can enable the DxBoardLink flow design configuration while creating a new DxDesigner project or after a project is created.

To enable the DxBoardLink flow design configuration when creating a new DxDesigner project, perform the following steps:

1. Start the DxDesigner software.

2. On the File menu, click **New** and click the **Project** tab. The **New** dialog box appears (Figure 6–20).

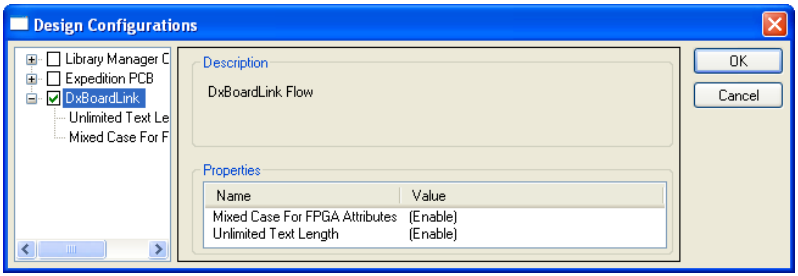
Figure 6–20. New Project Dialog Box



3. Click **More**. Turn on **DxBoardLink** (Figure 6–20).

To enable the DxBoardLink Flow design configuration in an existing project, click **Design Configurations** in the Design Configuration toolbar and turn on **DxBoardLink** (Figure 6–21).

Figure 6–21. DxBoardLink Design Configuration



DxDesigner Symbol Wizard

In addition to circuit simulation, circuit board schematic creation is one of the first tasks required in the design of a new PCB. Schematics are required to understand how the PCB will work, and to generate a netlist that is passed on to a board layout tool for board stackup design and routing.

You can create schematic symbols using the DxDesigner software based on FPGA designs exported from the Quartus II software through the Pin-Out file for instantiation in DxDesigner schematic design files. Most FPGA devices are physically large with hundreds of pins, requiring large schematic symbols that may not fit on a single schematic page. You can split or fracture symbols created in the DxDesigner software into a number of functional blocks, allowing multiple part fractures on the same schematic page or across multiple pages. In the DxDesigner software, these part fractures are joined together with the use of the `HETERO` attribute.

You can create schematic symbols in the DxDesigner software manually or with the Symbol Wizard. The DxDesigner Symbol Wizard is similar to the I/O Designer Symbol Wizard, but with fewer fracturing options.

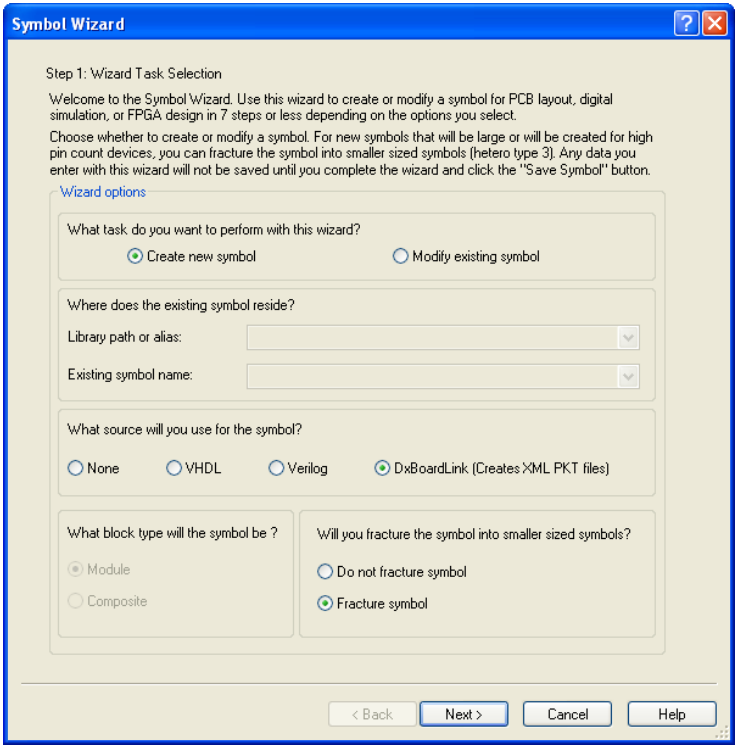
FPGA symbols based on Altera devices can be created, fractured, and edited using the DxDesigner Symbol Wizard. To start the Symbol Wizard, perform the following steps:

1. Start the DxDesigner software.

2. Click **Symbol Wizard** in the toolbar, or on the File menu, click **New**. The New window is shown. Click the **File** tab and create a new file of type **Symbol Wizard**.

3. Enter the new symbol name in the name field and click **OK**. The **Symbol Wizard** page is shown (Figure 6–22).

Figure 6–22. Wizard Task Selection



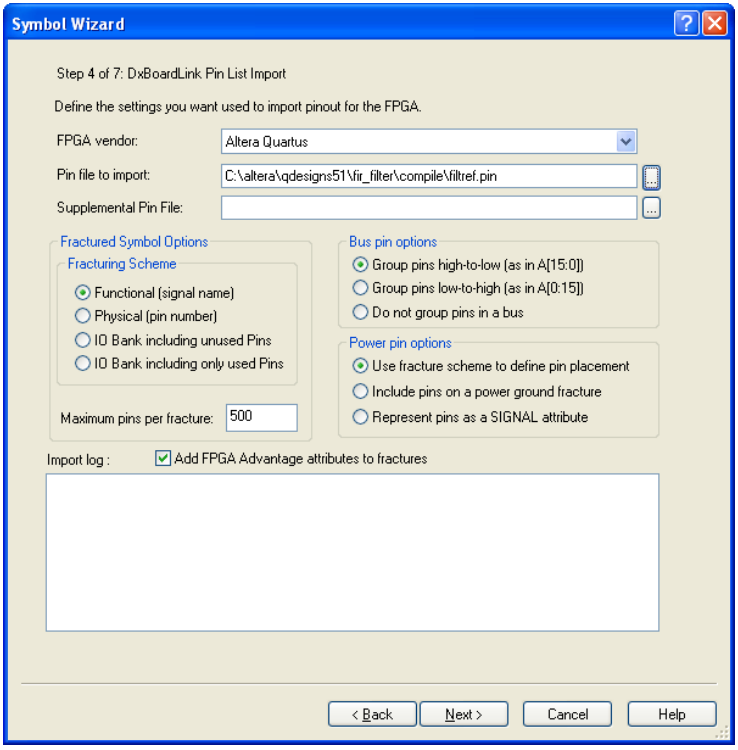
4. On the **Wizard Task Selection** page, choose to create a new symbol or modify an existing symbol. If you are modifying an existing symbol, specify the library path or alias, and select the existing symbol. If you are creating a new symbol, select DxBoardLink for the symbol source. The DxDesigner block type defaults to Module

because the FPGA design does not have an underlying DxDesigner schematic. Define whether or not to fracture the symbol. After making your selections, click **Next**. The **New Symbol and Library Name** page is shown.

5. On the **New Symbol and Library Name** page, enter a name for the symbol, an overall part name for all of the symbol fractures, and a library name for the new library created for this symbol. By default, the part and library names are the same as the symbol name. Click **Next**. The **Symbol Parameters** page is shown.

6. On the **Symbol Parameters** page, decide how the generated symbol will look and how it will match up with the grid you have set in your DxDesigner project schematic. After making your selections, click **Next**. The **DxBoardLink Pin List Import** page is shown (Figure 6–23).

Figure 6–23. DxBoardLink Pin List Import



7. On the **DxBoardLink Pin List Import** page, in the **FPGA vendor** list, select **Altera Quartus**. In the Pin-Out file to import field, browse to and select the Pin-Out file from your Quartus II design project directory. Additionally, select choices from the Fracturing Scheme options, Bus pin options, and Power pin options. After you make your selections, click **Next**. The **Symbol Attributes** page is shown.

8. On the **Symbol Attributes** page, select to create or modify symbol attributes for use in the DxDesigner software. After you make your selections, click **Next**. The **Pin Settings** page is shown.

9. On the **Pin Settings** page, make any final adjustments to pin and label location and information. Each tabbed spreadsheet represents a fracture of your symbol. After you make your selections, click **Save Symbol**.

After you save the symbol, you can examine and place any fracture of the symbol in your schematic. When you are finished with the Symbol Wizard, all the fractures you created are saved as separate files in the library you specified or created in the **/sym** directory in your DxDesigner project. You can add the symbols to your schematics or you can edit the symbols manually or with the Symbol Wizard.



Symbols created in the DxDesigner software can be edited and updated with newer versions of the Pin-Out file generated by the Quartus II software. However, symbol fracturing is fixed, and the symbol cannot be fractured again. To create new fractures for your design, create a new symbol in the Symbol Wizard, and follow the steps in "DxDesigner Symbol Wizard" on page 6–32.



For more information about creating, editing, and instantiating component symbols in DxDesigner, choose Schematic Design Help Topics from the Help menu in the DxDesigner software.

Conclusion

Transferring a complex, high-pin-count FPGA design to a PCB for prototyping or manufacturing is a daunting process that can lead to errors in the PCB netlist or design, especially when multiple engineers are working on different parts of the project. The design workflow available when the Quartus II software is used in conjunction with the Mentor Graphics toolset assists the FPGA designer and the board designer in preventing errors and focusing their attention on the design.

Referenced Documents

This chapter references the following documents:

- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Volume 3: Verification* of the *Quartus II Handbook*

Document Revision History

Table 6–2 shows the revision history for this chapter.

Table 6–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 6–36	—
May 2007 v7.1.0	• Removed table (6-2) listing of unsupported devices. • Added Referenced Documents.	—
March 2007 v7.0.0	Updated revision and publish date only.	—
November 2006 v6.1.0	Added revision history to the document.	—
May 2006 v6.0.0	Was chapter 7 in v5.1. Minor updates for the Quartus II software version 6.0.0.	—
November 2005 v5.1.1	Corrected text in steps 3 and 4 on page 11.	—
October 2005 v5.1.0	Initial release.	—

7. Cadence PCB Design Tools Support

QII52014-7.2.0

Introduction

With today's large, high-pin-count and high-speed FPGA devices, good printed circuit board (PCB) design practices are more essential than ever to ensure the correct operation of your system. Typically, the PCB design takes place concurrently with the design and programming of the FPGA. Signal and pin assignments are initially made by the FPGA or ASIC designer, and it is up to the board designer to correctly transfer these assignments to the symbols used in their system circuit schematics and board layout. As the board design progresses, pin reassignments may be requested or required to optimize the layout. These reassignments must in turn be relayed to the FPGA designer so that the new assignments can be processed through the FPGA using updated place-and-route.

Cadence provides tools to support this type of design flow. This chapter addresses how the Quartus® II software interacts with the Cadence Allegro Design Entry HDL software and the Allegro Design Entry CIS (Component Information System) software (also known as OrCAD Capture CIS) to provide a complete FPGA-to-board integration design workflow. This chapter provides information about the following topics:

- Cadence tool description, history, and comparison
- The general design flow between the Quartus II software and the Cadence Allegro Design Entry HDL software and the Cadence Allegro Design Entry CIS software
- Generating schematic symbols from your FPGA design for use in the Cadence Allegro Design Entry HDL software
- Updating Design Entry HDL symbols when signal and pin assignment changes are made in the Quartus II software
- Creating schematic symbols in the Cadence Allegro Design Entry CIS software from your FPGA design
- Updating symbols in the Cadence Allegro Design Entry CIS software when signal and pin assignment changes are made in the Quartus II software
- Using Altera®-provided device libraries in the Cadence Allegro Design Entry CIS software

This chapter is intended primarily for board design and layout engineers who want to begin the FPGA board integration process while the FPGA is still in the design phase. In addition, part librarians benefit from learning how to take output from the Quartus II software and use it to create new library parts and symbols.

The instructions in this chapter require the following software:

- The Quartus II software version 5.1 or later
- The Cadence Allegro Design Entry HDL or the Cadence Allegro Design Entry CIS software version 15.2 or later
- If you are using the OrCAD Capture software, you must have version 10.3 or later (CIS is optional)

Because the Cadence Allegro Design Entry CIS software is based on OrCAD Capture, these programs are very similar. For this reason, this chapter refers to the Allegro Design Entry CIS software in directions; however, these directions also apply to OrCAD Capture unless otherwise noted.

To obtain and license the Cadence tools described in this chapter, and for product information, support, and training, refer to the Cadence website, www.cadence.com. For information about OrCAD Capture and the CIS option, refer to the OrCAD website, www.orcad.com. For Cadence and OrCAD support and training, refer to the EMA Design Automation website, www.ema-eda.com.

Product Comparison

The design tools described in this chapter have similar functionality, but there are differences in their use as well as where to access product information. Table 7–1 lists the products described in this chapter and provides information about changes, product information, and support.

Table 7–1. Cadence and OrCAD Product Comparison

	Cadence Allegro Design Entry HDL	Cadence Allegro Design Entry CIS	OrCAD Capture CIS
Former Name	Concept HDL Expert	Capture CIS Studio	N/A
History	More commonly known by its former name, Cadence renamed all board design tools in 2004 under the Allegro name.	Based directly on OrCAD Capture CIS, this tool is still developed by OrCAD but sold and marketed by Cadence. EMA provides support and training.	The basis for Design Entry CIS is still developed by OrCAD for continued use by existing OrCAD customers. EMA now provides support and training for all OrCAD products.
Vendor Design Flow	Cadence Allegro 600 series, formerly known as Expert Series, for high-end, high-speed design.	Cadence Allegro 200 series, formerly known as Studio Series, for small- to medium-level design.	N/A
Information and Support	www.cadence.com www.ema-eda.com	www.cadence.com www.ema-eda.com www.orcad.com	www.ema-eda.com www.orcad.com

FPGA-to-PCB Design Flow

In the examples in this section, you create a design flow integrating an Altera FPGA design from the Quartus II software through a circuit schematic in the Allegro Design Entry HDL software (Figure 7–1) or the Allegro Design Entry CIS software (Figure 7–2).

Figure 7–1. Design Flow with the Allegro Design Entry HDL Software

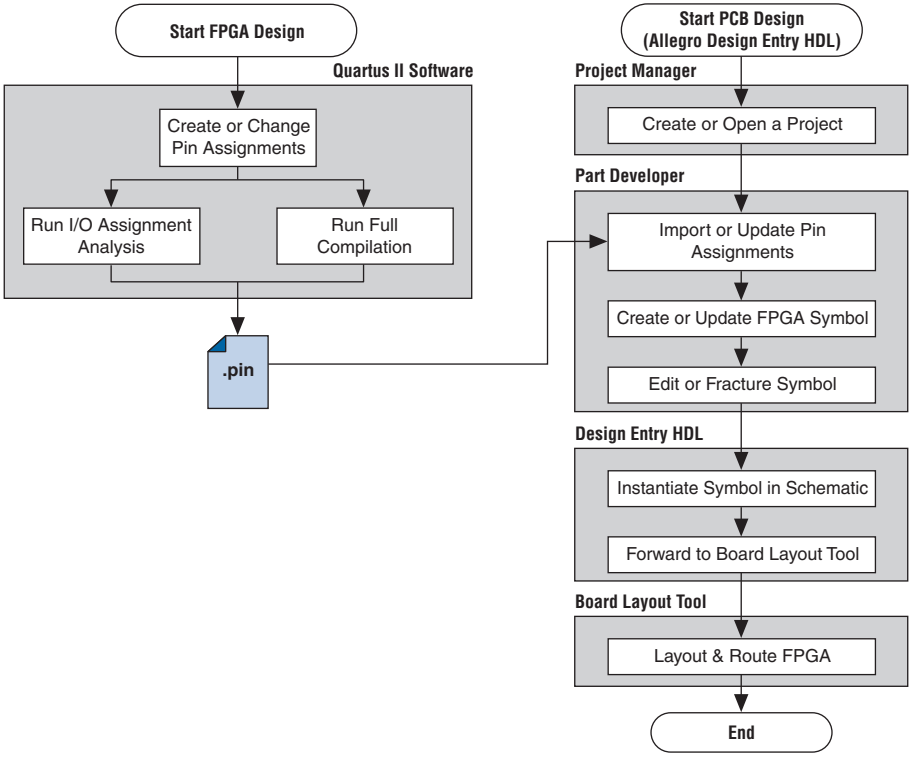
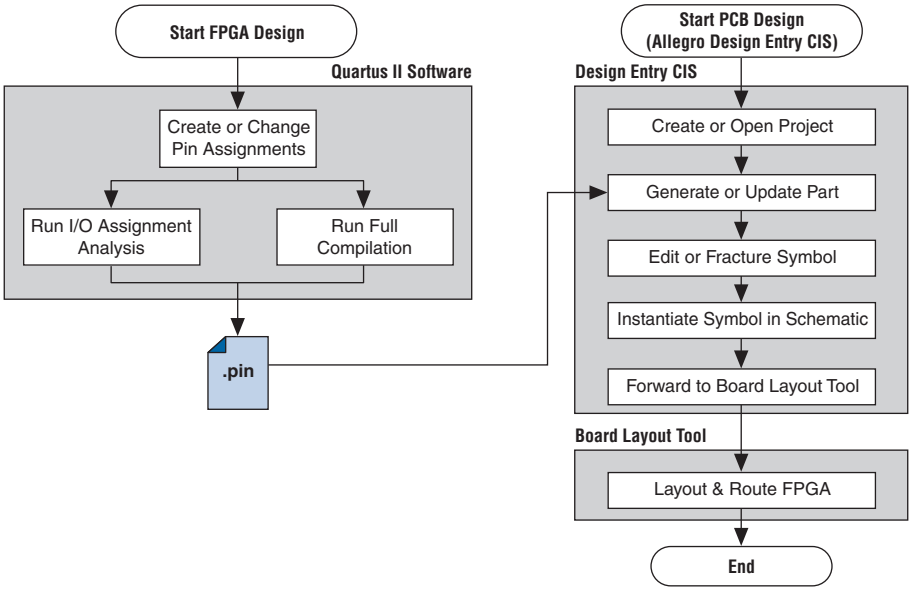


Figure 7–2. Design Flow with the Allegro Design Entry CIS Software



The basic steps in a complete design flow to integrate an Altera FPGA design starting in the Quartus II software through to a circuit schematic in Design Entry HDL or Design Entry CIS are as follows:

- Start the Quartus II software.
- In the Quartus II software, compile your design to generate a Pin-Out (**.pin**) file to transfer assignments to the Cadence tool.
- If you are using the Cadence Allegro Design Entry HDL software for your schematic design:
  - Open an existing project or create a new project in the Allegro Project Manager.
  - Construct a new symbol or update an existing symbol using the Allegro PCB Librarian Part Developer.
  - With the Part Developer, edit your symbol or fracture it into smaller parts, if desired.
  - Instantiate the symbol in your Design Entry HDL software schematic and transfer the design to your board layout tool.

- If you are using the Cadence Allegro Design Entry CIS software for your schematic design, perform the following steps:
  - Generate a new part within an existing or new Allegro Design Entry CIS project, referencing the Pin-Out file output from the Quartus II software. You can update an existing symbol with a new Pin-Out file.
  - Split the symbol into smaller parts as desired.
  - Instantiate the symbol in your Design Entry CIS schematic and transfer the design to your board layout tool.

Figures 7–1 and 7–2 show the possible design flows, depending on your tool choice. The Cadence PCB Librarian Expert license is required to use the PCB Librarian Part Developer to create FPGA symbols. You can update symbols with changes made to the FPGA design at any point using any of these tools.

Setting Up the Quartus II Software

You can transfer pin and signal assignments from the Quartus II software to the Cadence design tools by generating the Quartus II project Pin-Out file. The Pin-Out file is an output file generated by the Quartus II Fitter that contains pin assignment information. Use the Quartus II Pin Planner or Assignment Editor to set and change the assignments contained in the Pin-Out file. This file cannot be used to import pin assignment changes into the Quartus II software. Use it only to transfer assignments for use with the Cadence design tools.

The Pin-Out file lists all used and unused pins on your selected Altera device. It also provides the following basic information fields for each assigned pin on a device:

- Pin signal name and usage
- Pin number
- Signal direction
- I/O standard
- Voltage
- I/O bank
- User or Fitter-assigned



For information about using the Quartus II Pin Planner to create or change pin assignment details, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Generating Pin-Out Files

The Quartus II software automatically generates the Pin-Out file when your FPGA design is fully compiled or when you start I/O Assignment Analysis. To start I/O Assignment Analysis, on the Processing menu, point to Start and click **Start I/O Assignment Analysis**. The file is output by the Quartus II Fitter. The file is generated and placed in your Quartus II design directory with the name <*project name*>**.pin**. The Cadence design tools do not generate or change this file.

For more information about pin and signal assignment transfer and the files that the Quartus II software can import and export, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

FPGA-to-Board Integration with the Cadence Allegro Design Entry HDL Software

The Cadence Allegro Design Entry HDL software is Cadence's high-end schematic capture tool (part of the Cadence 600 series design flow). Use this software to create flat circuit schematics for all types of PCB design. The Cadence Allegro Design Entry HDL software can also create hierarchical schematics to facilitate design reuse and team-based design. With the Cadence Allegro Design Entry HDL software, the design flow from FPGA-to-board is one-way, using only the Pin-Out file generated by the Quartus II software. Signal and pin assignment changes can only be made in the Quartus II software and are reflected in updated symbols in a Design Entry HDL project.

Routing or pin assignment changes made in a board layout tool or a Design Entry HDL symbol cannot be back-annotated to the Quartus II software.

Figure 7–1 shows the design flow with the Cadence Allegro Design Entry HDL software.

For more information about the Cadence Allegro Design Entry HDL software and the Part Developer, including licensing, support, usage, training, and product updates, refer to the Help in the software or refer to the Cadence webpage at **www.cadence.com**.

Symbol Creation

In addition to circuit simulation, circuit board schematic creation is one of the first tasks required in the design of a new PCB. Schematics are required to understand how the PCB works, and to generate a netlist that is passed on to a board layout tool for board design and routing. The Allegro PCB Librarian Part Developer provides the ability to create schematic symbols based on FPGA designs exported from the Quartus II software.

Create symbols for Design Entry HDL with the Allegro PCB Librarian Part Developer available in the Allegro Project Manager. The Part Developer is the recommended method for importing FPGA designs into the Cadence Allegro Design Entry HDL software.

You must have a PCB Librarian Expert license from Cadence to run the Part Developer. The Part Developer provides a graphical interface with many options for creating, editing, fracturing, and updating symbols. If you do not use the Part Developer, you must create and edit symbols manually in the Symbol Schematic View in the Cadence Allegro Design Entry HDL software.



If you do not have a PCB Librarian Expert license, you can still automatically create FPGA symbols using the programmable IC (PIC) design flow found in the Allegro Project Manager. For more information about using the PIC design flow, refer to the Help in the Cadence design tools, or go to the Cadence website at **www.cadence.com**.

Before you create a symbol from an FPGA design, you must open or create a Design Entry HDL design project. You can do this with the Allegro Project Manager, the main interface to all of the Cadence tools.

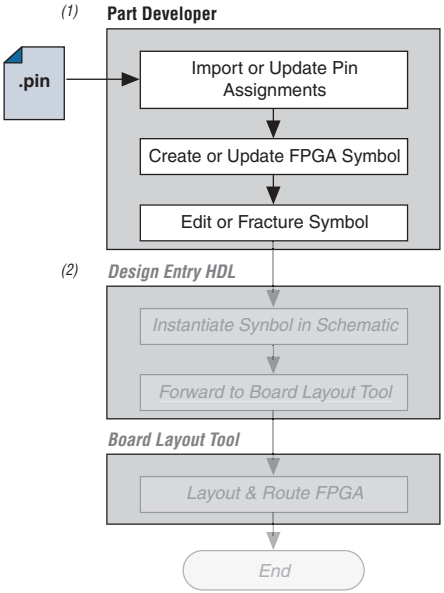
To open an existing design in the Allegro Project Manager, on the File menu, click **Open** and select the main design file for your project (found in your Allegro Design Entry HDL project directory and called <*project directory*>**.cpm**).

To create a new project, on the File menu, point to New and click **New Design**. The New Project Wizard appears. Use the wizard to name your new project, set the file location, and define associated part libraries.

Allegro PCB Librarian Part Developer

Create, fracture, and edit schematic symbols for your FPGA designs in Altera devices using the Part Developer. Most FPGA devices are physically large with hundreds of pins, requiring large schematic symbols that may not fit on a single schematic page. Symbols designed in the Part Developer can be split or fractured into a number of functional blocks called slots, allowing multiple smaller part fractures to exist on the same schematic page or across multiple pages. Figure 7–3 highlights how the Part Developer fits into the design flow.

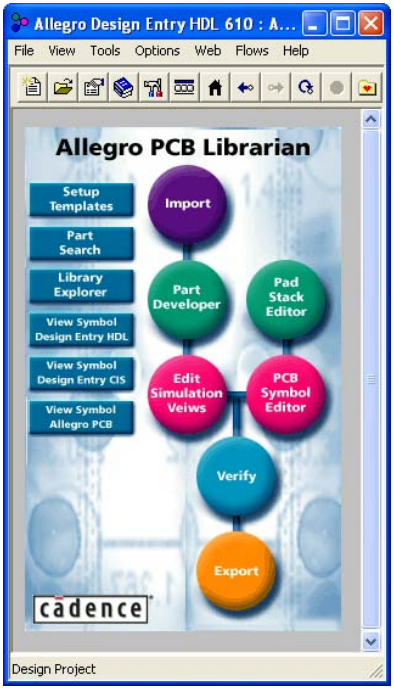
Figure 7–3. Part Developer in the Design Flow



Notes to Figure 7–3:
(1) Refer to Figure 7–1 for the full design flow flowchart details.
(2) Grayed out steps are not part of the FPGA Symbol creation or update process.

Run the Part Developer from the Project Manager (Figure 7–4). To start the Part Developer in the Project Manager, on the Flows menu, click **Library Management**. Click **Part Developer** to start the tool.

Figure 7–4. Invoking the Part Developer from the Project Manager

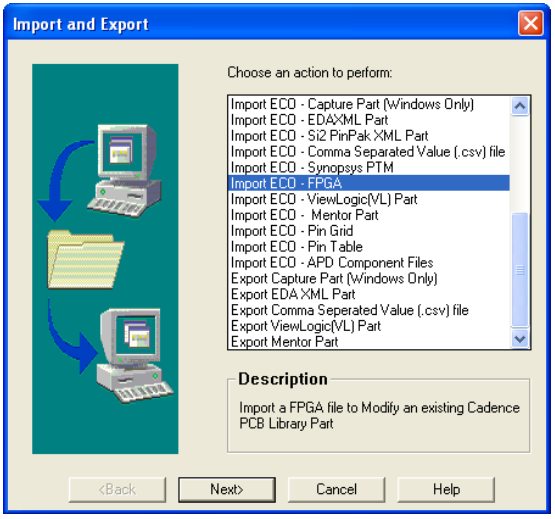


Import and Export Wizard

Once you are in the Part Developer, you can use the Import and Export Wizard to import your pin assignments from the Quartus II software. To access the Wizard, perform the following steps:

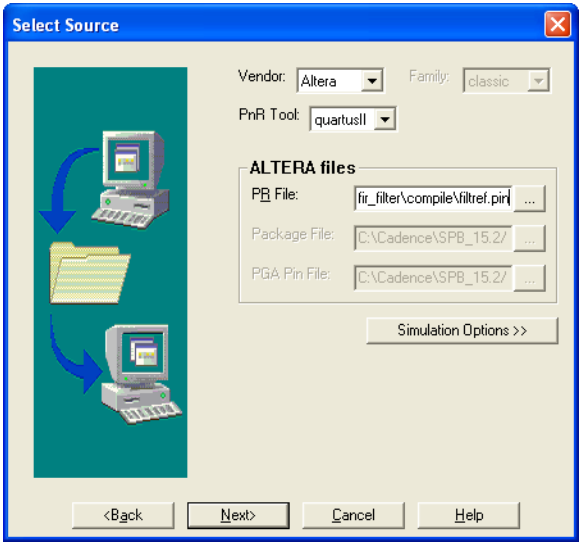
1. On the File menu, click **Import and Export**. The Import and Export Wizard appears (Figure 7–5).

Figure 7–5. Import and Export Wizard



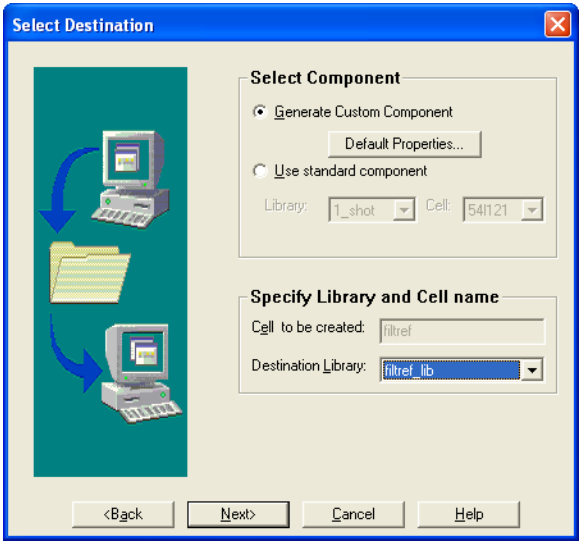
2. Select **Import FPGA**. Click **Next**. The **Select Source** page appears (Figure 7–6).

Figure 7–6. Select Source Page



3. In the **Vendor** list, select **Altera**. In the **PnR Tool** list, select **quartusII**. To specify the Pin-Out file in the **PR File** field, select the Pin-Out file in your Quartus II project directory. Click **Simulation Options** if you want to select simulation input files. Click **Next**. The **Select Destination** page is shown (Figure 7–7).

Figure 7–7. Select Destination Page



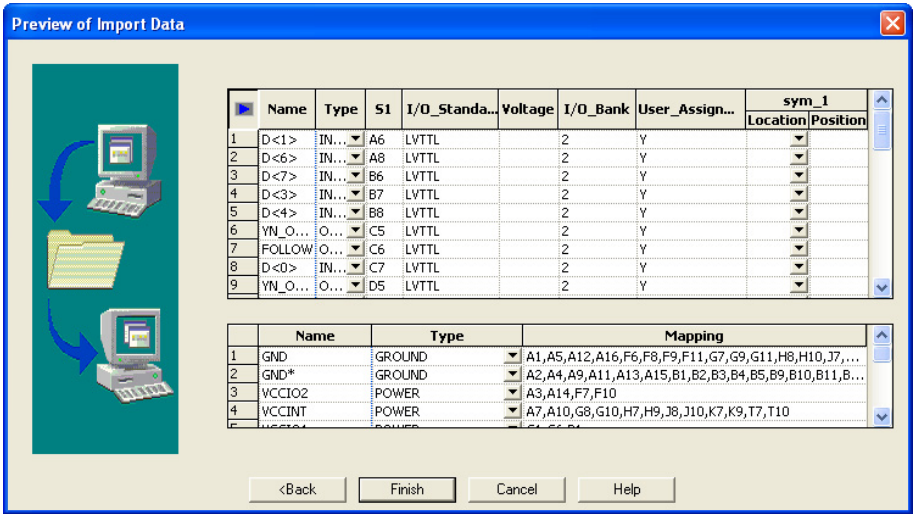
4. To create a new component in a library, click **Generate Custom Component**. To base your symbol on an existing component, click **Use standard component**.



You may want to do this if you previously created generic symbols for an FPGA device. You can place your pin and signal assignments from the Quartus II software on this symbol and reuse the symbol as a base any time you have a new FPGA design.

In the **Library** list, select an existing library. You can now select from the cells contained in the selected library. Each cell represents all of the symbol versions and part fractures for that particular part. In the **Cell** list, select the existing cell to use as a base for your part. In the **Destination Library** list, select a destination library for the component. Click **Next**. A preview of your import data is shown (Figure 7–8).

Figure 7–8. Preview of Import Data Window



5. Review the assignments you are importing into the Part Developer based on the data in the Pin-Out file. The location of each pin is not included in the information in this window, but inputs are placed on the left side of the created symbol, outputs on the right, power pins on the top, and ground pins on the bottom. Make any desired changes. When you have completed your changes, click **Finish** to create the symbol. The Part Developer main screen is shown.

If the Part Developer is not set up to point to your PCB Librarian Expert license file, an error message displays in red at the bottom of the message text window of the **Part Developer** when you select the Import and Export command. To point to your PCB Librarian Expert license, on the **File** menu, click **Change Product** and select the correct product license.

For more information about licensing and obtaining licensing support, contact Cadence or refer to their website at **www.cadence.com**.

Edit and Fracture Symbol

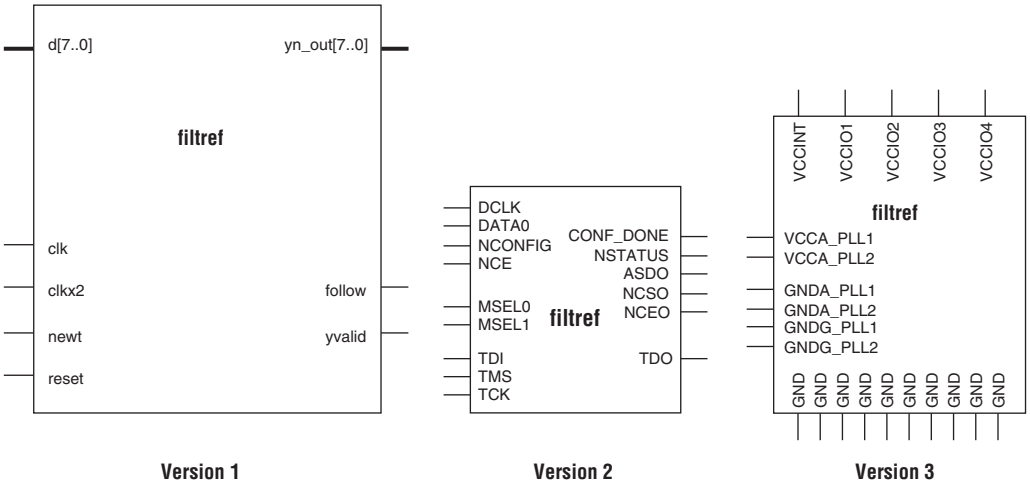
After you save your new symbol in the Part Developer software, you can edit the symbol graphics, fracture the symbol into multiple slots, and add or change package or symbol properties. These actions are available from the **Part Developer** main window.

The Part Developer Symbol Editor contains many graphical tools to edit the graphics of a particular symbol. Select the symbol in the cell hierarchy to edit the symbol graphics. The **Symbol Pins** tab is shown. Edit the preview graphic of the symbol in the **Symbol Pins** tab.

Fracturing a Part Developer package into separate symbol slots is especially useful for FPGA designs. A single symbol for most FPGA packages may be too large for a single schematic page. Splitting the part into separate slots allows you to organize parts of the symbol by function, creating cleaner circuit schematics. For example, you could create one slot for an I/O symbol, a second slot for a JTAG symbol, and a third slot for a power/ground symbol.

Figure 7–9 shows a part fractured into separate slots.

Figure 7–9. Splitting a Symbol into Multiple Slots *Notes (1), (2)*



Notes to Figure 7–9:

(1) Figure 7–9 represents a Cyclone device with JTAG or passive serial (PS) mode configuration option settings. Symbols created for other devices or other configuration modes may have different sets of configuration pins, but can be fractured in a similar manner.

(2) Symbol fractures are referred to in different ways in each of the tools described in this chapter. Refer to Table 7–2 for the specific tool naming conventions.

(3) The power/ground slot shows only a representation of power and ground pins. In actuality, the device contains a high number of power and ground pins.



While the Part Developer software refers to symbol fractures as slots, the other tools described in this chapter use different names to refer to symbol fractures. Table 7–2 lists the symbol fracture naming conventions for each of the tools addressed in this chapter.

Table 7–2. Symbol Fracture Naming

	Allegro PCB Librarian Part Developer Software	Allegro Design Entry HDL Software	Allegro Design Entry CIS Software
During symbol generation	Slots	N/A	Sections
During symbol schematic instantiation	N/A	Versions	Parts

To fracture a part into separate slots, or modify the slot locations of pins on parts that are already fractured in the Part Developer, perform the following steps:

1. Start the Cadence Allegro Design Project Manager.
2. On the Flows menu, click **Library Management**. The Library Management design flow is shown. Click **Part Developer**. The Part Developer launches.
3. Click on the name of the package you want to change in the cell hierarchy. The **Package Pin** tab appears.
4. Click **Functions/Slots**. If you are not creating new slots but want to change the slot location of some pins, proceed to step 5. If you are creating new slots, click **Add**. A dialog box appears, allowing you to add extra symbol slots. Set the number of extra slots you want to add to the existing symbol, not the total number of desired slots for the part. Click **OK**.
5. Click **Distribute Pins**. Set the slot where each pin should reside. Use the checkboxes in each column to move pins from one slot to another. You can use the standard cut, copy, and paste keyboard commands on selected groups of checkboxes to move multiple pins from one slot to another. Click **OK**.
6. After distributing the pins, click the **Package Pin** tab and click **Generate Symbol(s)**. the **Generate Symbols** dialog box appears.
7. Select whether to create a new symbol or modify an existing symbol in each slot. Click **OK**.

The newly generated or modified slot symbols display as separate symbols in the cell hierarchy. Each of these symbols can be edited individually.



The Part Developer lets you remap pin assignments in the Package Pin tab of the main **Part Developer** window. If signals are remapped to different pins in the Part Developer, the changes are reflected only in regenerated symbols for use in your schematics. You cannot transfer pin assignment changes to the Quartus II software from the Part Developer, which creates a potential mismatch of the schematic symbols and assignments in the FPGA design. If pin assignment changes are necessary, make the changes in the Quartus II Pin Planner instead of the Part Developer, and update the symbol as described in the following sections.

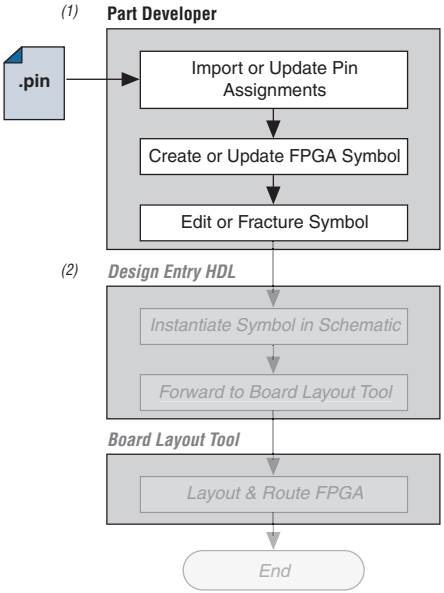


For more information about creating, editing, and organizing component symbols with the Allegro PCB Librarian Part Developer, refer to the Part Developer Help.

Update FPGA Symbol

As the design process continues, you may need to make changes to the logic design in the Quartus II software, placing signals on different pins after the design is recompiled, or use the Quartus II Pin Planner to make changes manually. The board designer may request such changes to improve the board routing and layout. These types of changes must be carried forward to the circuit schematic and board layout tools to ensure signals are connected to the correct pins on the FPGA. Updating the Pin-Out file in the Quartus II software facilitates this flow. Figure 7–10 shows this part of the design flow.

Figure 7–10. Updating the FPGA Symbol in the Design Flow



Notes to Figure 7–10:
(1) Refer to Figure 7–1 for the full design flow flowchart details.
(2) Grayed out steps are not part of the FPGA Symbol update process.

Once the Pin-Out file has been updated, perform the following steps to update the symbol using the Allegro PCB Librarian Part Developer:

1. On the File menu, click **Import and Export**. The Import and Export Wizard appears.

2. In the list of actions to perform, select **Import ECO - FPGA.** Click **Next**. The **Select Source Page** is shown.

3. Select the updated source of the FPGA assignment information. In the **Vendor** list, select **Altera**. In the **PnR Tool** list, select **quartusII**. In the **PR File** field, click browse to specify the updated Pin-Out file in your Quartus II project directory. Click **Next**. The Select Destination window is shown.

4. Select the source component and a destination cell for the updated symbol. To create a new component based on the updated pin assignment data, select **Generate Custom Component**. This replaces the cell listed under the Specify Library and Cell name header with a new, non-fractured cell. Any symbol edits or fractures

are lost. You can preserve these edits by selecting **Use standard component and select the existing library and cell**. Select the destination library for the component and click **Next**. The Preview of Import Data page is shown.

5. Make any additional changes to your symbol. Click **Next**. A list of ECO messages displays summarizing what changes will be made to the cell. To accept the changes and update the cell, click **Finish**.

6. The main **Part Developer** window is shown. You can edit, fracture, and generate the updated symbols as usual from this window.



If the Part Developer is not set up to point to your PCB Librarian Expert license file, an error message displays in red at the bottom of the message text window of the **Part Developer** when you select the **Import and Export** command. To point to your PCB Librarian Expert license, on the File menu, click **Change Product**, and select the correct product license. For more information about licensing and obtaining licensing support, contact Cadence or refer to their website at **www.cadence.com**.

Instantiating the Symbol in the Cadence Allegro Design Entry HDL Software

Once the new symbol is saved in the Part Developer, instantiate the symbol in your Design Entry HDL schematic.

1. In the Allegro Project Manager, switch to the board design flow.

2. On the Flows menu, click **Board Design**.

3. Click **Design Entry** to start the Design Entry HDL software.

4. To add the newly created symbol to your schematic, right-click in the main schematic window and choose **Add Component**, or on the Component menu, click **Add**. The **Add Component** dialog box appears.

5. Select the new symbol library location, and select the name of the cell you created from the list of cells.

The symbol is now "attached" to your cursor for placement in the schematic. If you fractured the symbol into slots, right-click the symbol and choose **Version** to select one of the slots for placement in the schematic.

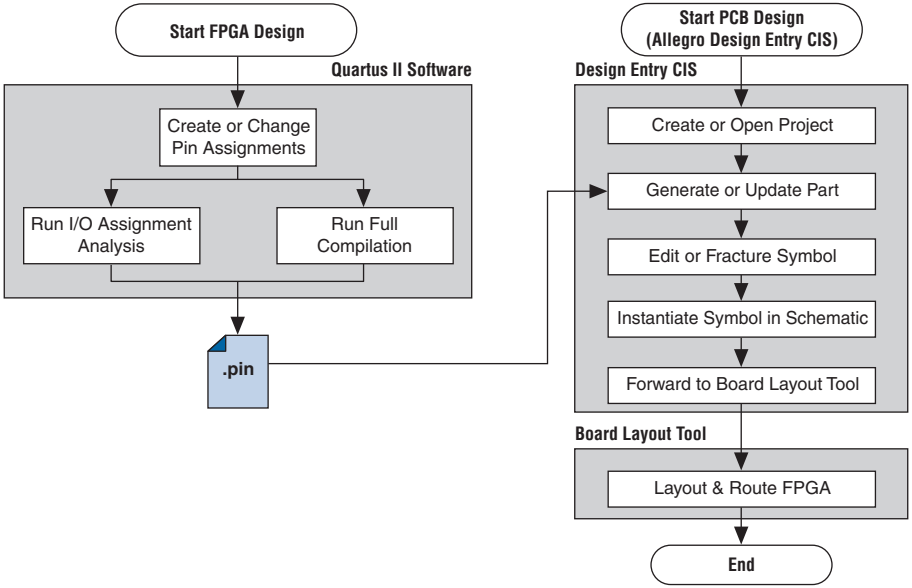
For more information about the Cadence Allegro Design Entry HDL software, including licensing, support, usage, training, and product updates, refer to the Help in the software or go to the Cadence website at **www.cadence.com**.

FPGA-to-Board Integration with Allegro Design Entry CIS

The Cadence Allegro Design Entry CIS software is Cadence's mid-level schematic capture tool (part of the Cadence 200 series design flow based on OrCAD Capture CIS). Use this software to create flat circuit schematics for all types of PCB design. You can also create hierarchical schematics to facilitate design reuse and team-based design using this software. With the Cadence Allegro Design Entry CIS software, the design flow from FPGA-to-board is unidirectional using only the Pin-Out file generated by the Quartus II software. Signal and pin assignment changes can only be made in the Quartus II software and are reflected in updated symbols in a Design Entry CIS schematic project.

Routing or pin assignment changes made in a board layout tool or a Design Entry CIS symbol cannot be back-annotated to the Quartus II software. Figure 7–11 shows the design flow with the Cadence Allegro Design Entry CIS software.

Figure 7–11. Design Flow with the Cadence Allegro Design Entry CIS Software





For more information about the Cadence Allegro Design Entry CIS software, including licensing, support, usage, training, and product updates, refer to the Help in the software, go to the Cadence website at **www.cadence.com**, or go to the EMA Design Automation website at **www.ema-eda.com**.

Allegro Design Entry CIS Project Creation

The Cadence Allegro Design Entry CIS software has built-in support for creating schematic symbols using pin assignment information imported from the Quartus II software.

If you have not already created a new project in the Cadence Allegro Design Entry CIS software, perform the following steps to create a new project:

1. On the File menu, point to **New** and click **Project**. The New Project Wizard starts.

   When you create a new project, you can select the PC Board Wizard, the Programmable Logic Wizard, or a blank schematic.

2. Select the **PC Board Wizard** to create a project where you can select which part libraries to use, or select a blank schematic.

   The Programmable Logic Wizard is used only to build an FPGA logic design in the Cadence Allegro Design Entry CIS software, which is unnecessary when using the Quartus II software.

   No other special configuration for your project is required. Your new project is created in the specified location and initially consists of two files: the OrCAD Capture Project (**.opj**) file and the Schematic Design (**.dsn**) file.

Generate Part

After you create a new project or open an existing project in the Allegro Design Entry CIS software, you can generate a new schematic symbol based on your Quartus II FPGA design. You can also update an existing symbol if your Pin-Out file has been updated in the Quartus II software. The Cadence Allegro Design Entry CIS software stores component symbols in OrCAD Library (**.olb**) files. When a symbol is placed in a library attached to a project, it is immediately available for instantiation in the project schematic.

You can add symbols to an existing library or you can create a new library specifically for the symbols generated from your FPGA designs. To create a new library, perform the following steps:

1. On the File menu, point to New and click **Library** in the Cadence Allegro Design Entry CIS software to create a default library named library1.olb. This library appears in the Library folder in the Project Manager window of the Cadence Allegro Design Entry CIS software.

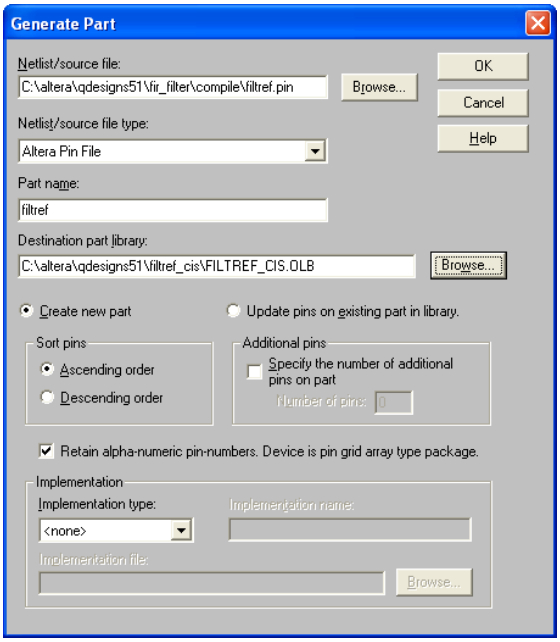
2. Right-click the new library and choose **Save As** to specify a desired name and location for the library. The library file is not created until you save the new library.

You can now create a new symbol to represent your FPGA design in your schematic. To generate a schematic symbol, perform the following steps:

1. Start the Cadence Allegro Design Entry CIS software.

2. On the Tools menu, click **Generate Part**. The **Generate Part** dialog box appears (Figure 7–12).

Figure 7–12. Generate Part Dialog Box



3. In the **Netlist/source file type** field, click **Browse** to specify the Pin-Out file from your Quartus II design.

4. In the **Netlist/source file type** list, select **Altera Pin File**.

5. Enter the new part name.

6. Specify the Destination part library for the symbol. If you do not select an existing library for the part, a new library is created with a default name that matches the name of your Design Entry CIS project.

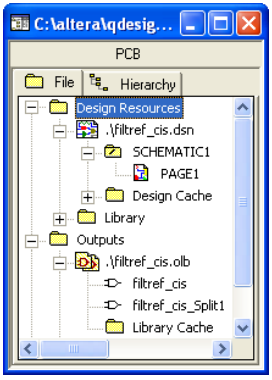
7. Select **Create new part** if you are creating a brand new symbol for this design. Select **Update pins on existing part in library** if you updated your Pin-Out file in the Quartus II software and want to transfer any assignment changes to an existing symbol.

8. Select any other desired options and set Implementation type to **<none>**. The symbol is for a primitive library part based only on the Pin-Out file and does not need a special implementation. Click **OK**.

9. Review the Undo warning and click **Yes** to complete the symbol generation.

The symbol is generated and placed in the selected library or in a new library found in the Outputs folder of the design in the Project Manager window (Figure 7–13). Double-click the name of the new symbol to see its graphical representation and edit it manually using the tools available in the Cadence Allegro Design Entry CIS software.

Figure 7–13. Project Manager Window



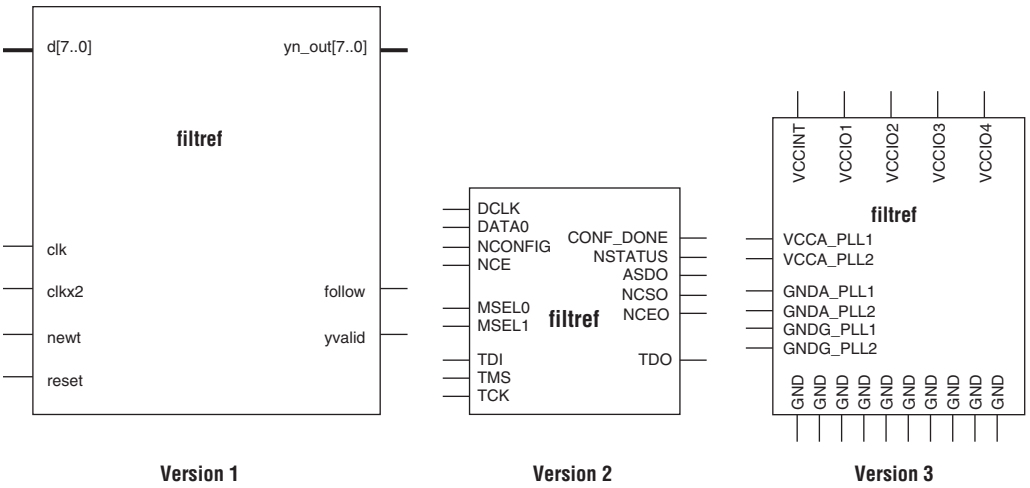
For more information about creating and editing symbols in the Allegro Design Entry CIS software, refer to the Help in the software.

Split Part

Once a new symbol is saved in a project's library, you can fracture the symbol into multiple parts called sections. Fracturing a part into separate sections is especially useful for FPGA designs. A single symbol for most FPGA packages may be too large for a single schematic page. Splitting the part into separate sections allows you to organize parts of the symbol by function, creating cleaner circuit schematics. For example, you could create one slot for an I/O symbol, a second slot for a JTAG symbol, and a third slot for a power/ground symbol.

Figure 7–14 shows a part fractured into separate sections.

Figure 7–14. Splitting a Symbol into Multiple Sections *Notes (1), (2)*



Notes to Figure 7–14:

(1) Figure 7–14 represents a Cyclone device with JTAG or passive serial (PS) mode configuration option settings. Symbols created for other devices or other configuration modes may have different sets of configuration pins, but can be fractured in a similar manner.

(2) Symbol fractures are referred to in different ways in each of the tools described in this chapter. Refer to Table 7–2 for the specific tool naming conventions.

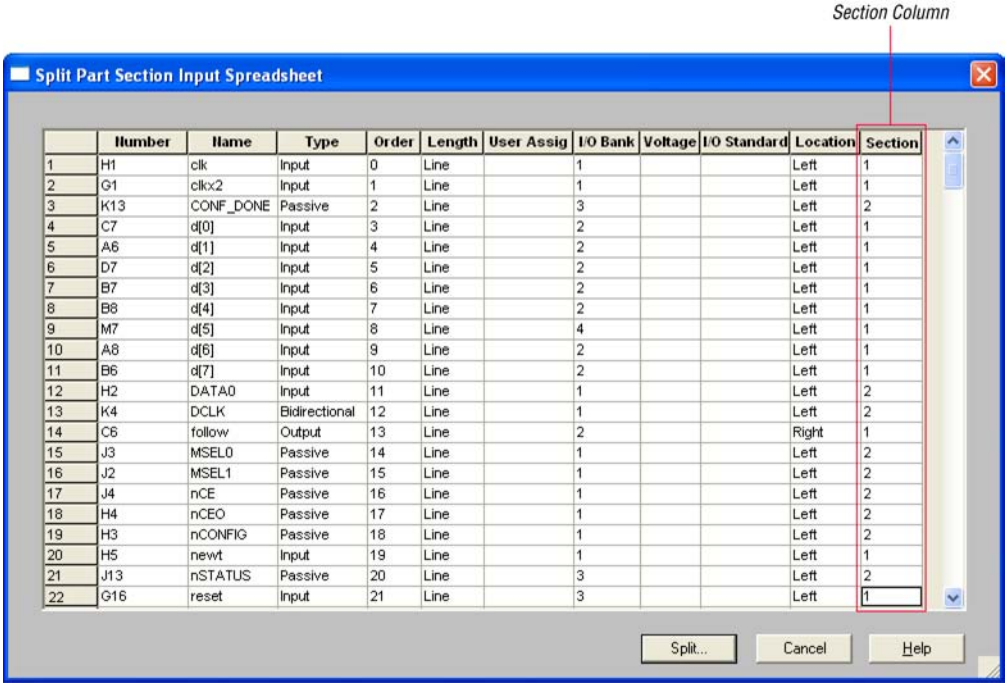
(3) The power/ground section shows only a representation of power and ground pins. In actuality, the device contains a high number of power and ground pins.



While symbol generation in the Design Entry CIS software refers to symbol fractures as sections, the other tools described in this chapter use different names to refer to symbol fractures. Refer to Table 7–2 on page 7–14 for the symbol fracture naming conventions for each of the tools addressed in this chapter.

To split a part into sections, select the part in its library in the Project Manager window of Design Entry CIS. On the Tools menu, click **Split Part** or right-click the part and choose **Split Part**. The **Split Part Section Input Spreadsheet** is shown (Figure 7–15).

Figure 7–15. Split Part Section Input Spreadsheet



	Number	Name	Type	Order	Length	User Assig	I/O Bank	Voltage	I/O Standard	Location	Section
1	H1	clk	Input	0	Line		1			Left	1
2	G1	clkx2	Input	1	Line		1			Left	1
3	K13	CONF_DONE	Passive	2	Line		3			Left	2
4	C7	d[0]	Input	3	Line		2			Left	1
5	A6	d[1]	Input	4	Line		2			Left	1
6	D7	d[2]	Input	5	Line		2			Left	1
7	B7	d[3]	Input	6	Line		2			Left	1
8	B8	d[4]	Input	7	Line		2			Left	1
9	M7	d[5]	Input	8	Line		4			Left	1
10	A8	d[6]	Input	9	Line		2			Left	1
11	B6	d[7]	Input	10	Line		2			Left	1
12	H2	DATA0	Input	11	Line		1			Left	2
13	K4	DCLK	Bidirectional	12	Line		1			Left	2
14	C6	follow	Output	13	Line		2			Right	1
15	J3	MSEL0	Passive	14	Line		1			Left	2
16	J2	MSEL1	Passive	15	Line		1			Left	2
17	J4	nCE	Passive	16	Line		1			Left	2
18	H4	nCEO	Passive	17	Line		1			Left	2
19	H3	nCONFIG	Passive	18	Line		1			Left	2
20	H5	newt	Input	19	Line		1			Left	1
21	J13	nSTATUS	Passive	20	Line		3			Left	2
22	G16	reset	Input	21	Line		3			Left	1

Each row in the spreadsheet represents a pin in the symbol. The spreadsheet column labeled Section indicates the section of the symbol to which each pin is assigned. By default, all pins in a new symbol are located in section 1. Change the values in this column to assign pins to different, new sections of the symbol. You can also specify the side of a section on which the pin will reside by changing the values in the Location column. When you are finished, click **Split**. A new symbol appears in the same library as the original with the name *<original part name>*_**Split1.**

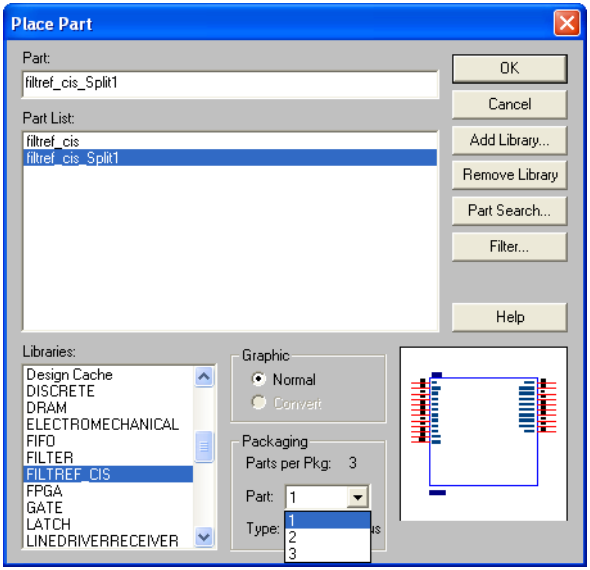
View and edit each section individually. To view the new sections of the part, double-click the part. The Part Symbol Editor window is shown. The first section of the part is displayed for editing. On the View menu, click **Package** to view thumbnails of all the part sections. Double-click a thumbnail to edit that section of the symbol.

For more information about splitting parts into sections and editing symbol sections in the Cadence Allegro Design Entry CIS software, refer to the Help in the software.

Instantiate Symbol in Design Entry CIS Schematic

After a new symbol is saved in a library in your Design Entry CIS project, you can instantiate it on a page in your schematic. Open a schematic page in the Project Manager window of the Cadence Allegro Design Entry CIS software. On the schematic page, to add the newly created symbol to your schematic, on the Place menu, click **Part**. The **Place Part** dialog box appears (Figure 7–16).

Figure 7–16. Place Part Dialog Box



Select the new symbol library location and the newly created part name. If you select a part that is split into sections, you can select the section to place from the Part pop-up menu. Click **OK**. The symbol is now attached to your cursor for placement in the schematic. Click on the schematic page to place the symbol.



For more information about using the Cadence Allegro Design Entry CIS software, refer to the Help in the software.

Altera Libraries for Design Entry CIS

Altera provides downloadable OrCAD Library Files for many of its device packages. You can add these libraries to your Design Entry CIS project and update the symbols with the pin assignments contained in the Pin-Out file generated by the Quartus II software. This allows you to use the downloaded library symbols as a base for creating custom schematic symbols with your pin assignments that you can edit or fracture as desired. This can increase productivity by reducing the amount of time it takes to create and edit a new symbol.

To use the Altera-provided libraries with your Design Entry CIS project, perform the following steps:

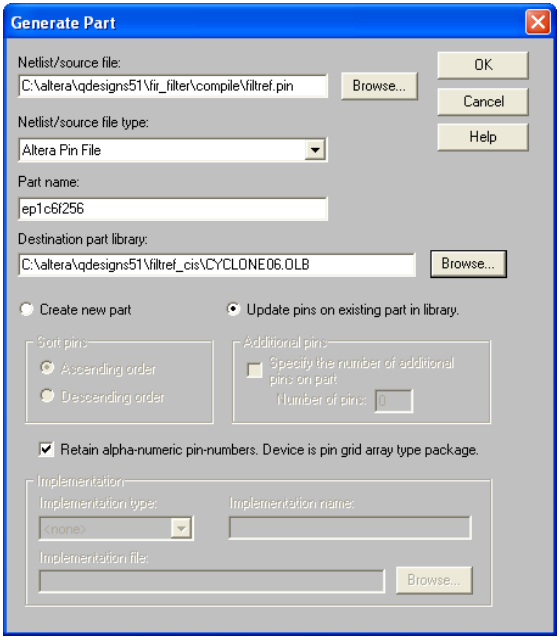
1. Download the library of your target device from the Download Center page found through the Support page on the Altera website at **www.altera.com**.

2. Make a copy of the appropriate OrCAD Library file so that the original symbols are not altered. Place the copy in a convenient location such as your Design Entry CIS project directory.

3. In the Project Manager window of the Cadence Allegro Design Entry CIS software, click once on the Library folder to select it. On the Edit menu, click **Project** or right-click the Library folder and choose **Add File** to select the copy of the downloaded OrCAD Library file and add it to your project. The new library is added to the list of part libraries for your project.

4. On the Tools menu, click **Generate Part.** The **Generate Part** dialog box appears (Figure 7–17).

Figure 7–17. Generate Part Dialog Box



5. In the **Netlist/source file type** field, click **Browse** to specify the Pin-Out file in your Quartus II design.

6. From the **Netlist/source file type** list, select **Altera Pin File**.

7. For the part name, enter the name of the target device the same as it appears in the downloaded library file. For example, if you are using a device from the CYCLONE06.OLB library, set the part name to match one of the devices in this library such as ep1c6f256. You can rename the symbol later in the Project Manager window after the part is updated.

8. Set the Destination part library to the copy of the downloaded library you added to the project.

9. Select **Update pins on existing part in library**. Click **OK,** then click **Yes**.

The symbol is updated with your pin assignments. Double-click the symbol in the Project Manager window to view and edit the symbol. On the View menu, click **Package** if you want to view and edit other sections

of the symbol. If the symbol in the downloaded library is already fractured into sections, as some of the larger packages are, you can edit each section but you cannot further fracture the part. Generate a new part without using the downloaded part library if you require additional sections.



For more information about creating, editing, and fracturing symbols in the Cadence Allegro Design Entry CIS software, refer to the Help in the software.

Conclusion

Transferring a complex, high-pin-count FPGA design to a PCB for prototyping or manufacturing is a daunting process that can lead to errors in the PCB netlist or design, especially when different engineers are working on different parts of the project. The design workflow available when the Quartus II software is used with tools from Cadence assists the FPGA designer and the board designer in preventing such errors and focusing all attention on the design.

Referenced Document

This chapter references the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Document Revision History

Table 7–3 shows the revision history for this chapter.

Table 7–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No changes to the chapter.	—
May 2007 v7.1.0	Added Referenced Document on page 7–27.	—
March 2007 v7.0.0	Updated revision and publish date only.	—
November 2006 v6.1.0	Added revision history to the document.	—
May 2006 v6.0.0	Was chapter 6 in v5.1. Minor updates for the Quartus II software version 6.0.0.	—
November 2005 v5.1.1	Realigned figures 6-9 and 6-14.	—
October 2005 v5.1.0	Initial release.	—

Section III. Area, Timing and Power Optimization

Techniques for achieving the highest design performance are important when designing for programmable logic devices (PLDs), especially higher density FPGAs. The Altera® Quartus® II software offers a number of features to help you optimize your design. The software also includes advanced tools that allow for detailed analysis of your design, including fully integrated floorplan tools that allow you to easily determine and locate critical paths in the targeted device floorplan. This section explains how to use these tools and options to enhance your FPGA design analysis flow.

This section includes the following chapters:

- Chapter 8, Area and Timing Optimization
- Chapter 9, Power Optimization
- Chapter 10, Analyzing and Optimizing the Design Floorplan
- Chapter 11, Netlist Optimizations and Physical Synthesis
- Chapter 12, Design Space Explorer
- Chapter 13, Synplicity Amplify Physical Synthesis Support



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

8. Area and Timing Optimization

QII52005-7.2.0

Introduction

Good optimization techniques are essential for achieving the highest possible quality of results when designing for programmable logic devices (PLDs). The optimization features available in the Quartus® II software are designed to allow you to meet design requirements by applying these techniques at multiple points in the design process.

This chapter explains techniques to reduce resource usage, improve timing performance, and reduce compilation times when designing for Altera® devices. It also explains how and when to use some of the features described in other chapters of the *Quartus II Handbook*. This introduction describes the various stages in a design optimization process, and points you to the appropriate sections in the chapter for area, timing, or compilation time optimization.

Topics in this chapter include:

- "Initial Compilation: Required Settings" on page 8–3
- "Initial Compilation: Optional Settings" on page 8–8
- "Design Analysis" on page 8–14
- "Resource Utilization Optimization Techniques (LUT-Based Devices)" on page 8–25
- "Timing Optimization Techniques (LUT-Based Devices)" on page 8–42
- "Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)" on page 8–71
- "Timing Optimization Techniques (Macrocell-Based CPLDs)" on page 8–79
- "Compilation-Time Optimization Techniques" on page 8–86
- "Other Optimizing Resources" on page 8–93
- "Scripting Support" on page 8–94

The applicability of these techniques varies from design to design. Applying each technique does not always improve design results. Settings and options in the Quartus II software have default values that generally provide the best trade-off between compilation time, resource utilization, and timing performance. You can adjust these settings to determine whether other settings provide better results for your design.

When using advanced optimization settings and tools, it is important to benchmark their effect on your quality of results and to use them only if they improve results for your design.

Use the optimization flow described in this chapter to explore various compiler settings and determine the techniques that provide the best results.

Optimizing Your Design

The first stage in the optimization process is to perform an initial compilation to view the quality of results for your design. "Initial Compilation: Required Settings" on page 8–3 provides guidelines on some of the settings and assignments that are recommended for your initial compilation. "Initial Compilation: Optional Settings" on page 8–8 gives you a set of settings that you may have to turn on based on your design requirements. "Design Analysis" on page 8–14 explains how to analyze the compilation results.



The incremental compilation methodology can be used as a part of the optimization process. Even though incremental compilation alone may not reduce the resource usage, it can be used as a tool for timing preservation and compilation time reduction when used in conjunction with other techniques. Incremental compilation can be used as a tool to attain timing closure.



For more details about the incremental compilation methodology with the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

After you have analyzed the compilation results, perform the optimization stages in the recommended order, as described in this chapter.

For LUT-based devices (FPGAs and MAX® II CPLDs), perform optimizations in the following order:

1. If your design does not fit, refer to "Resource Utilization Optimization Techniques (LUT-Based Devices)" on page 8–25 before trying to optimize I/O timing or register-to-register timing.

2. If your design does not meet the required I/O timing performance, refer to "I/O Timing Optimization Techniques (LUT-Based Devices)" on page 8–97 before trying to optimize register-to-register timing.

3. If your design does not meet the required slack on any of the clock domains in the design, refer to "Register-to-Register Timing Optimization Techniques (LUT-Based Devices)" on page 8–52.

For macrocell-based devices (MAX 7000 and MAX 3000 CPLDs), perform optimizations in the following order:

1. If your design does not fit, refer to "Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)" on page 8–71 before trying to optimize I/O timing or register-to-register timing.

2. If your timing performance requirements are not met, refer to "Timing Optimization Techniques (Macrocell-Based CPLDs)" on page 8–79.

For device-independent techniques to reduce compilation time, refer to "Compilation-Time Optimization Techniques" on page 8–86.

You can use all these techniques in the GUI or with Tcl commands. For more information about scripting techniques, refer to "Scripting Support" on page 8–94.

Initial Compilation: Required Settings

This section describes the basic assignments and settings for your initial compilation. Ensure that you check all the following suggested compilation assignments before compiling the design in the Quartus II software. Significantly different compilation results can occur depending on the assignments you have set.

The following settings are required:

- "Device Settings" on page 8–3
- "I/O Assignments" on page 8–4
- "Timing Requirement Settings" on page 8–4
- "Device Migration Settings" on page 8–7
- "Partitions and Floorplan Assignments for Incremental Compilation" on page 8–7

Device Settings

Assigning a specific device determines the timing model that the Quartus II software uses during compilation. Choose the correct speed grade to obtain accurate results and the best optimization. The device size and the package determine the device pin-out and how many resources are available in the device.

To choose the target device, on the Assignments menu, click **Device**.

In a Tcl script, use the following command to set the device.

```
set_global_ssignment -name DEVICE <device>
```

I/O Assignments

The I/O standards and drive strengths specified for a design affect I/O timing. Specify I/O assignments so that the Quartus II software uses accurate I/O timing delays in timing analysis and Fitter optimizations.

The Quartus II software can choose pin locations automatically for best quality of results. If your pin locations are not fixed due to printed circuit board (PCB) layout requirements, leave pin locations unconstrained to achieve the best results. If your pin locations are already fixed, make pin assignments to constrain the compilation appropriately. "Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)" on page 8–71 includes recommendations for making pin assignments that can have a larger effect on your quality of results in smaller macrocell-based architectures.

Use the **Assignment Editor** and **Pin Planner** to assign I/O standards and pin locations.

For more information about I/O standards and pin constraints, refer to the appropriate handbook. For information about planning and checking I/O assignments, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*. For information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

Timing Requirement Settings

An important step in the optimal quality of results, especially for high-performance FPGA designs, is to make comprehensive timing requirement settings. It is important to apply these settings for the following reasons:

- Correct timing assignments allow the software to work hardest to optimize the performance of the timing-critical parts of the design, and make trade-offs for performance. This optimization can also save area or power utilization in non-critical parts of the design.
- If enabled, the Quartus II software performs physical synthesis optimizations based on timing requirements (refer to "Synthesis Netlist Optimizations and Physical Synthesis Optimizations" on page 8–54 for more information).

- Depending on the Fitter Effort setting, the Quartus II Fitter may reduce runtime considerably if your timing requirements are being met. For a full description of the different effort levels, refer to "Fitter Effort Setting" on page 8–13.

As a general rule, do not over-constrain the software by applying timing requirements that are higher than your design requirements. Use your real design requirements to get the best results. Power utilization may also be larger in an over-constrained design, when the software balances power and performance during compilation.



In some designs with multiple clocks, it may be possible to improve the timing performance on one clock domain while reducing the performance on other clock domains by over-constraining the most important clock. If you use this technique, ensure that any performance improvements that you see are real gains by performing a sweep over multiple seeds. For more information, refer to "Fitter Seed" on page 8–62.

When you are over-constraining one of the clocks, Altera recommends that you use the Auto Fit option rather than the Standard Fit option in the Fitter settings. This helps to reduce the compilation time.

The Auto Fit option may increase the number of routing wires used. This can lead to an increase in the dynamic power, when compared to using the Standard Fit option, unless **Extra effort PowerPlay power optimization** is also enabled. When you turn on **Extra effort PowerPlay power optimization**, Auto Fit continues to optimize for reduction of wire usage even after meeting the register-to-register requirement. There is no adverse effect on the dynamic power consumption. If dynamic power consumption is a concern, make sure to set the PowerPlay power optimization to Extra effort in both the Analysis and Synthesis settings and the Fitter settings.



For more details, refer to the *Power Driven Compilation* section in the *Power Optimization* chapter of the *Quartus II Handbook*.

- The Timing Analyzer (Classic or TimeQuest) checks your design against the timing assignments. The compilation report and timing analysis reporting commands show whether timing requirements are met, and provide detailed timing information about paths that violate timing requirements.

To make clock assignments for the Quartus II Classic Timing Analyzer, on the Assignments menu, click **Timing Analysis Settings**. Select the **Classic Timing Analyzer Settings** page. Use the **Delay requirements**, **Minimum delay requirements**, and **Clock Settings** boxes to make global

settings, or to apply settings to individual clocks, click **Individual Clocks** (recommended for multiple-clock designs). Create the clock setting, and apply it to the appropriate clock node in the design. The Timing Wizard can also step you through the process of making individual clock constraints for the Quartus II Classic Timing Analyzer. To run the Timing Wizard, on the Assignments menu, click **Timing Wizard**.

To make clock and timing assignments for the Quartus II TimeQuest Timing Analyzer, create a Synopsys Design Constraint (**.sdc**) file that contains all of your constraints. You can also create constraints in the TimeQuest GUI. Use the `write_sdc` command, or, in the TimeQuest Timing Analyzer, on the Constraints menu, click **Write SDC File** to write your constraints to an SDC file. You can add an SDC file to your project on the **TimeQuest Timing Analyzer** page under **Timing Analysis Settings**.

Ensure that every clock signal has an accurate clock setting assignment. If clocks come from a common oscillator, they may be considered related. Ensure that all related or derived clocks are set up correctly in the assignments. All I/O pins that require I/O timing optimization must have settings. You should also specify minimum timing constraints as applicable. If there is more than one clock or there are different I/O requirements for different pins, make multiple clock settings and individual I/O assignments instead of using the global settings.

Make any complex timing assignments required in the design, including any cut-timing and multicycle path assignments. Common situations for these types of assignments include reset or static control signals, cases in which it is not important how long it takes a signal to reach a destination, and paths that can operate in more than one clock cycle. These assignments allow the Quartus II software to make appropriate trade-offs between timing paths, and can enable the Compiler to improve timing performance in other parts of the design. Specify these settings in the Assignment Editor.



For more information about timing assignments and timing analysis, refer to the *Quartus II Classic Timing Analyzer* and the *Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*.

Timing Constraint Check—Report Unconstrained Paths

To ensure that all constraints or assignments have been applied to design nodes, you can report all unconstrained paths in your design.

While using the Quartus II TimeQuest Timing Analyzer, you can report all the unconstrained paths in your design with the Report Unconstrained Paths command (`report_ucp`) in the Task pane.

If you are using the Quartus II Classic Timing Analyzer, on the Assignments menu, click **Timing Analysis Settings**. In the **Settings** dialog box, under Timing Analysis Settings, select **Classic Timing Analyzer Settings**. Click **More Settings**. The **More Timing Settings** dialog box appears. From the **Existing option settings** list, select **Report Unconstrained Paths**. From the Setting drop-down list, select **On**.

Device Migration Settings

If you anticipate a change to the target device later in the design cycle either because of changes in the design or other considerations, plan for it at the beginning of your design cycle. Whenever you select a target device in the **Settings** page from the **Assignments** menu, you can also list any other compatible devices you can migrate to by clicking on the **Migration Devices** tab in the **Settings** page. If you plan to move your design to a HardCopy device, make sure to select the device from the pull-down menu under the **Companion device** tab in the **Settings** page.

Selecting the migration device and companion device early in the design cycle helps you to minimize changes to the design at a later stage.

Partitions and Floorplan Assignments for Incremental Compilation

The Quartus II incremental compilation feature enables hierarchical and team-based design flows in which you compile parts of your design while other parts of the design remain unchanged, or import parts of your design from separate Quartus II projects.

Using the incremental compilation methodology with a good partitioning of the design can often help to achieve timing closure. Creating LogicLock regions and using incremental compilation can help you achieve timing closure block by block, preserve the timing performance between iterations, and therefore help achieve timing closure for the entire design.

Using incremental compilation also helps reduce compilation times. You can analyze the critical paths in the Chip Planner, then make changes in the floorplan to meet timing requirements with the Incremental Design methodology.

For information about using the incremental compilation feature to reduce your compilation time, refer to "Incremental Compilation" on page 8–86.

If you want to take advantage of this feature for a team-based design flow, to reduce your compilation times, or to improve the timing performance of your design during iterative compilation runs, make meaningful design partitions as well as create a floorplan for your design partitions. Assignments can negatively affect a design's quality of results if you do not follow Altera's recommendations. Good assignments can improve your quality of results.



If you plan to use incremental compilation, you must create a floorplan for your design. If you are not using incremental compilation, this step is optional.



For guidelines about how to create partition and floorplan assignments for your design, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Initial Compilation: Optional Settings

This section describes the settings that are optional, but which may be helpful for compiling your design. You can selectively set all the optional settings that help to improve performance (if required) and reduce compilation time. These settings vary between designs, and there is no standard set that applies to all designs. Significantly different compilation results can occur depending on the assignments you have set.

The following settings are optional:

- "Design Assistant" on page 8–9
- "Smart Compilation Setting" on page 8–9
- "Early Timing Estimation" on page 8–9
- "Optimize Hold Timing" on page 8–10
- "Asynchronous Control Signal Recovery/Removal Analysis" on page 8–11
- "Limit to One Fitting Attempt" on page 8–12

Design Assistant

You can run the Design Assistant to analyze the post-fitting results of your design during a full compilation. The Design Assistant checks rules related to areas such as gated clocks, reset signals, asynchronous design practices, and signal race conditions. This is especially useful during the early stages of your design, so that you can work on any areas of concern in your design before proceeding with design optimization.

On the Assignments menu, click **Settings**. In the **Category** list, select **Design Assistant** and turn on **Run Design Assistant during compilation**.

You can also specify which rules you want the Design Assistant to apply when analyzing and generating messages for a design.



For more information about the rules in the design assistant, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

Smart Compilation Setting

Smart compilation can reduce compilation time by skipping compiler stages that are not needed to recompile the design. This is especially useful when you perform multiple compilation iterations during the optimization phase of the design process. However, smart compilation uses more disk space. To turn on smart compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings** and turn on **Use Smart Compilation**.



This feature skips entire compiler stages (such as Analysis and Synthesis) when they are not needed. This feature is different from incremental compilation, which you can use to compile parts of your design while preserving results for unchanged parts. For information about using the incremental compilation feature to reduce your compilation time, refer to "Incremental Compilation" on page 8–86.

Early Timing Estimation

The Quartus II software provides an Early Timing Estimation feature that estimates your design's timing results before the software performs full placement and routing. On the Processing menu, point to Start, and click **Start Early Timing Estimate** to generate initial compilation results after you have run analysis and synthesis. When you want a quick estimate of a design's performance before proceeding with further design or synthesis tasks, this command can save significant compilation time.

Using this feature provides a timing estimate up to 45× faster than running a full compilation, and the fit is not fully optimized or routed. Therefore, the timing report is only an estimate. On average, the estimated delays are within 11% of those achieved by a full compilation compared to the final timing results.

You can specify what type of delay estimates to use with this feature. On the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings**, and select **Early Timing Estimate**. On the **Early Timing Estimate** page, the following options are available:

- The **Realistic** option, which is the default, generates delay estimates that will likely be close to the results of a full compilation.
- The **Optimistic** option uses delay estimates that are lower than those likely to be achieved by a full compilation, which results in an optimistic performance estimate.
- The **Pessimistic** option uses delay estimates that are higher than those likely to be achieved by a full compilation, which results in a pessimistic performance estimate.

All three options offer the same reduction in compilation time.

You can use the Chip Planner or the Timing Closure Floorplan (for older devices) to view the placement estimate created by this feature to identify critical paths in the design. Then, if necessary, you can add or modify floorplan constraints such as LogicLock™ regions, or make other changes to the design. You can then rerun the Early Timing Estimator to quickly assess the impact of any floorplan assignments or logic changes, enabling you to try different design variations and find the best solution.

Optimize Hold Timing

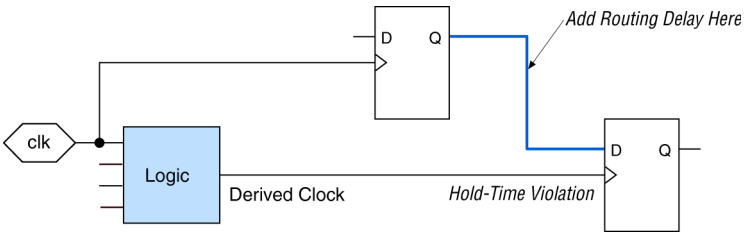
The **Optimize Hold Timing** option directs the Quartus II software to optimize minimum delay timing constraints. This option is available only for the Arria™ GX devices, Stratix® series of devices, Cyclone® series of devices, and MAX II devices. When you turn on this option, the Quartus II software adds delay to connections to guarantee that the minimum delay requirements are satisfied.

In the **Fitter Settings** panel, if you choose **I/O Paths and Minimum TPD Paths** (the default choice if you turn on **Optimize Hold Timing**), the Fitter works to meet the following criteria:

- Hold times (t_H) from device input pins to registers
- Minimum delays from I/O pins to I/O registers or from I/O registers to I/O pins
- Minimum clock-to-out time (t_{CO}) from registers to output pins

If you select **All paths**, the Fitter also works to meet hold requirements from registers to registers, as in Figure 8–1, where a derived clock generated with logic causes a hold time problem on another register. However, if your design has internal hold time violations between registers, Altera recommends that you correct the problems by making changes to your design, such as using a clock enable signal instead of a derived or gated clock.

Figure 8–1. Optimize Hold Timing Option Fixing an Internal Hold Time Violation



For design practices that can help eliminate internal hold time violations, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

Asynchronous Control Signal Recovery/Removal Analysis

This option determines whether you want the software to analyze the results of recovery and removal checks for paths that end at an asynchronous clear, preset, or load signal of a register. Recovery time is the minimum length of time an asynchronous control signal, for example, clear and preset, must be stable before the active clock edge. Removal time is the minimum time an asynchronous control signal must be stable after the active clock edge. Recovery and removal requirements are similar to setup and hold time requirements, respectively.

When using the Quartus II Classic Timing Analyzer for timing analysis, Recovery/Removal analysis is turned off by default. Turning on the option adds additional constraints during placement and routing which can increase compilation time and reduce performance. If this analysis is required, on the Assignments menu, click **Settings**. In the **Category** list, select **Timing Requirements & Options**, then click **More Settings**. Turn on **Enable Recovery/Removal analysis**. By running the Recovery/Removal analysis, you can make sure that no timing failures are related to the asynchronous control signals in your design.

When using TimeQuest for timing analysis, Recovery/Removal analysis and optimization are always performed during placement and routing. You can use the `create_timing_summary` Tcl command to report the recovery and removal analysis. The slack for Removal/Recovery is determined in a similar way to Setup and Hold checks.



For more details about Recovery/Removal analysis with the TimeQuest Timing Analyzer, refer to *The Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

For designs containing FIFOs, Altera recommends turning on Recovery/Removal analysis. Recovery/Removal analysis helps to analyze corner-case conditions to achieve better functional coverage.

Limit to One Fitting Attempt

A design may fail to fit for several reasons, such as logic overuse or illegal assignments. For most of these failures, the Quartus II software informs you of the problem. However, if the design uses too much routing, the Quartus II software makes up to two additional attempts to fit your design, each time trying harder than the last to reduce routing resource utilization. Each of these fit attempts takes significantly longer than the previous attempt.

However, for large designs, you may not wish to wait for all three attempts, and would rather receive an error message earlier. This would allow you to take corrective action sooner.

To have the software indicate whether the design fails to fit after the first attempt, in the Fitter settings, check **Limit to One Fitting Attempt**.

Refer to "Routing" on page 8–38 for instructions about how to lower the design's routing utilization, so your design may be made to fit into the target device if it fails to fit due to the lack of routing resources.

Optimize Fast Corner Timing

Historically, FPGA timing analysis has been performed using only worst-case delays, which are described in the slow corner timing model. However, due to process variation and changes in the operating conditions, delays on some paths can be significantly smaller than those in the slow corner timing model. This can result in hold time violations on those paths, and in rare cases, additional setup time violations.

By default, the Fitter optimizes constraints using only the slow corner timing model. You can turn on the Optimize Fast Corner Timing setting to instruct the Fitter to also optimize constraints using the fast corner timing model to avoid the possible timing violations described above.

To turn on the Optimize Fast Corner Timing setting, from the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings** and turn on **Optimize Fast Corner Timing**. Using the two different timing models can be important to account for process, voltage, and temperature variations for each device. Turning this option on increases compilation time by approximately 10%.

For designs with DDR interfaces (that interface to external memories), make sure you turn on the Optimize Fast Corner Timing switch, so that your timing analysis results are accurate.

Fitter Effort Setting

On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**. The Fitter effort settings are **Auto Fit**, **Standard Fit**, and **Fast Fit**. The default setting depends on the device family specified.

Auto Fit

The **Auto Fit** option (available only for Arria GX devices, the Stratix and Cyclone series of devices, and MAX II devices) focuses full Fitter effort on those aspects of the design that require further optimization. Auto Fit can significantly reduce compilation time relative to Standard Fit if your design has some easy-to-meet timing requirements, low routing resource utilization, or both. However, those designs that require full optimization generally receive the same effort as is achieved by selecting Standard Fit. Auto Fit is the default Fitter effort setting for all devices for which this option is available.

If you want the Fitter to attempt to exceed the timing requirements by a certain margin instead of simply meeting them, specify a minimum slack in the **Desired worst-case slack** box.



Specifying a minimum slack does not guarantee that the Fitter achieves the slack requirement; it only guarantees that the Fitter applies full optimization unless the target slack is exceeded.

There may be cases in which your design has one or two clock domains on which you want to maximize the slack, while on the other clock domains you want to meet the timing without any requirement of maximizing the slack. Over-constraining the

clock on which you need to maximize the slack, while using the Auto Fit option, increases the chances that the Fitter is able to meet this requirement.

The Auto Fit option also causes the Quartus II Fitter to optimize for shorter compilation times instead of maximum performance if the design includes no timing assignments. For designs with no timing assignments, the resulting f_{MAX} is, on average 10% lower than that achieved using the Standard Fit option. If your design has aggressive timing requirements or is hard to route, the placement does not stop early, and the compilation time is the same as using the Standard Fit option. For designs with no timing requirements or easily achieved timing requirements, you can achieve an average compilation time reduction of 40% by using the Auto Fit option.

Standard Fit

Use the **Standard Fit** option to exceed specified timing requirements and achieve the best possible timing results and lowest routing resource utilization for your design. However, this setting usually increases compilation time relative to Auto Fit, because it applies full optimization, regardless of the design requirement.

Fast Fit

The **Fast Fit** option reduces the amount of optimization effort for each algorithm employed during fitting. This option reduces the compilation time by about 50%, resulting in a fit that has, on average, 10% lower f_{MAX} than that achieved using the **Standard Fit** setting. For a small fraction of hard-to-fit circuits, the reduced optimization that results from using the **Fast Fit** option can cause the first fitting attempt to fail due to routing problems, resulting in multiple fitting attempts and increased compilation time.

You can also set the Fitter Effort using the following Tcl command in your script:

```
set_global_assignment  -name FITTER_EFFORT <value>
```

In this case, <*value*> can be `AUTO_FIT`, `FAST_FIT`, or `STANDARD_FIT`.

Design Analysis

The initial compilation establishes whether the design achieves a successful fit and meets the specified performance. This section describes how to analyze your design results in the Quartus II software. After design analysis, proceed to optimization as described in "Optimizing Your Design" on page 8–2.

Error and Warning Messages

After your initial compilation, it is important to evaluate all error and warning messages to see if any design or setting changes are required. If needed, make these changes and recompile the design before proceeding with design optimization.

To suppress messages that you have evaluated and that can be ignored, right-click on the message in the Messages window and click **Suppress**.



For more information about message suppression, refer to the *Message Suppression* section in the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

Ignored Timing Assignments

You can use the Ignored Timings Assignments page in the Compilation Report to view any assignments that were ignored by the Quartus II Classic Timing Analyzer during the previous compilation. The Quartus II Classic Timing Analyzer ignores assignments that are invalid, conflict with other assignments, or become obsolete through the use of other assignments. If any assignments are ignored, analyze why they were ignored. If necessary, correct the assignments and recompile the design before proceeding with design optimization.

If you are using TimeQuest for timing analysis, you can use the following command to generate a listing of ignored timing constraints:

```
report_sdc -ignored -panel_name "Ignored Constraints"
```



For more information about the `report_sdc` command and its options, refer to *The Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Resource Utilization

Determining device utilization is important regardless of whether a successful fit is achieved. If your compilation results in a no-fit error, resource utilization information is important for analyzing the fitting problems in your design. If your fitting is successful, review the resource utilization information to determine whether the future addition of extra logic or other design changes may introduce fitting difficulties.

To determine resource usage, refer to the **Flow Summary** section of the Compilation Report. This section reports how many resources are used, including pins, memory bits, digital signal processing (DSP) block 9-bit elements (for Arria GX, Stratix, and Stratix II devices) or 18-bit elements

(for Stratix III devices), and phase-locked loops (PLLs). The **Flow Summary** indicates whether the design exceeds the available device resources. More detailed information is available by viewing the reports under **Resource Section** in the **Fitter** section of the **Compilation Report**.



For Arria GX, Stratix II, and Stratix III devices, a device with low utilization does not have the lowest adaptive logic module (ALM) utilization possible. For these devices, the Fitter uses adaptive look-up tables (ALUTs) in different ALMs—even when the logic can be placed within one ALM—to achieve the best timing and routability results. In achieving these results, the Fitter may spread logic throughout the device. As the device fills up, the Fitter automatically searches for logic functions with common inputs to place in one ALM. The number of partnered ALUTs and packed registers also increases. Therefore, a design that is reported as close to 100% full might still have space for extra logic if logic and registers can be packed together more aggressively.

If resource usage is reported as less than 100% and a successful fit cannot be achieved, either there are not enough routing resources or some assignments are illegal. In either case, a message appears in the **Processing** tab of the Messages window describing the problem.

If the Fitter finishes very quickly compared to Fitter runs on similar designs, then a resource might be over-utilized or there might be an illegal assignment. If the Quartus II software seems to run for an excessively long time compared to runs on similar designs, then a legal placement or route probably cannot be found. Look for errors and warnings that indicate these types of problems.

Refer to "Limit to One Fitting Attempt" on page 8–12 for more information about how to get a quick error message on hard-to-fit designs.

You can use the Chip Planner or the Timing Closure Floorplan (for older devices) to find areas of the device that have routing congestion.



For details about using the Chip Planner and the Timing Closure Floorplan tools, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

I/O Timing (Including t_{PD})

The Quartus II TimeQuest Timing Analyzer supports the Synopsys Design Constraints (SDC) format for constraining your design. When using TimeQuest for timing analysis, use the `set_input_delay`

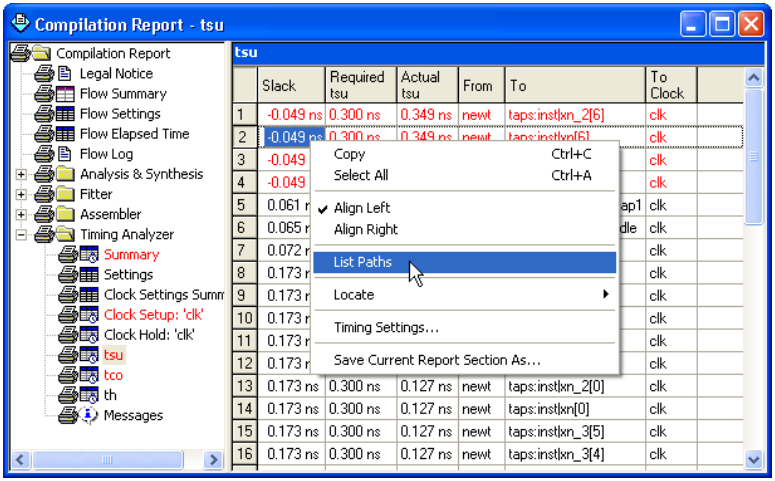
constraint to specify the data arrival time at an input port with respect to a given clock. For output ports, use the `set_output_delay` command with respect to a given clock. You can use the `report_timing` Tcl command to generate the required I/O timing reports.

The rest of this section refers to timing settings and analysis in the Quartus II Classic Timing Analyzer. For more details about equivalent settings and analysis in the Quartus II TimeQuest Timing Analyzer, refer to *The Quartus II TimeQuest Timing Analyzer* and *Switching to the Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*.

If you are using the Quartus II Classic Timing Analyzer, from the Compilation Report, select the **Timing Analyzer** to determine whether or not I/O timing has been met. The t_{SU}, t_H, and t_{CO} reports list the I/O paths, together with the required timing number if you have specified a timing requirement, the actual timing number for the timing as reported by the Quartus II software, and the slack (the difference between your requirement and the actual number). If you have any point-to-point propagation delay (t_{PD}) assignments, the t_{PD} report lists the corresponding paths.

The I/O paths that do not meet the required timing performance are reported as having negative slack and are displayed in red (Figure 8–2). In cases when you do not make an explicit I/O timing assignment to an I/O pin, the Quartus II timing analysis software still reports the **Actual** number, which is the timing number that must be met for that timing parameter when the device runs in your system.

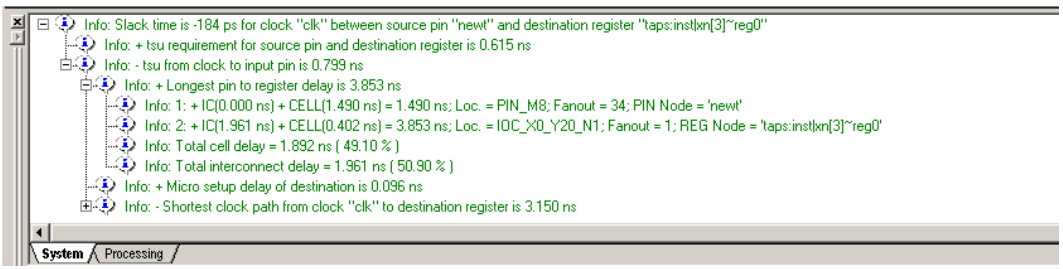
Figure 8–2. I/O Timing Analyzer Report



To analyze the reasons why your timing requirements are not met, right-click an entry in the report and click **List Paths** (Figure 8–2). A message listing the paths appears in the **System** tab of the Messages window. To expand a selection, click the icon at the beginning of the line (Figure 8–3). This is a good method to determine where the greatest delay is located along the path.

The List Paths report lists the slack time and how that slack time was calculated. By expanding the various entries, you can see the incremental delay through each node in the path as well as the total delay. The incremental delay is the sum of the interconnect delay (IC) and the cell delay (CELL) through the logic.

Figure 8–3. I/O Slack Report



To analyze I/O timing, right-click on an I/O entry in the report, point to Locate, and click **Locate in Chip Planner** or **Timing Closure Floorplan** (for older devices) to highlight the I/O path on the floorplan. Negative slack indicates paths that failed to meet their timing requirements. Other options allow you to see all the intermediate nodes (combinational logic cells) on a path and the delay for each level of logic, or to see the fan-in and fan-out of a selected node.



For more information about how timing numbers are calculated, refer to the *Quartus II Classic Timing Analyzer* chapter or the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

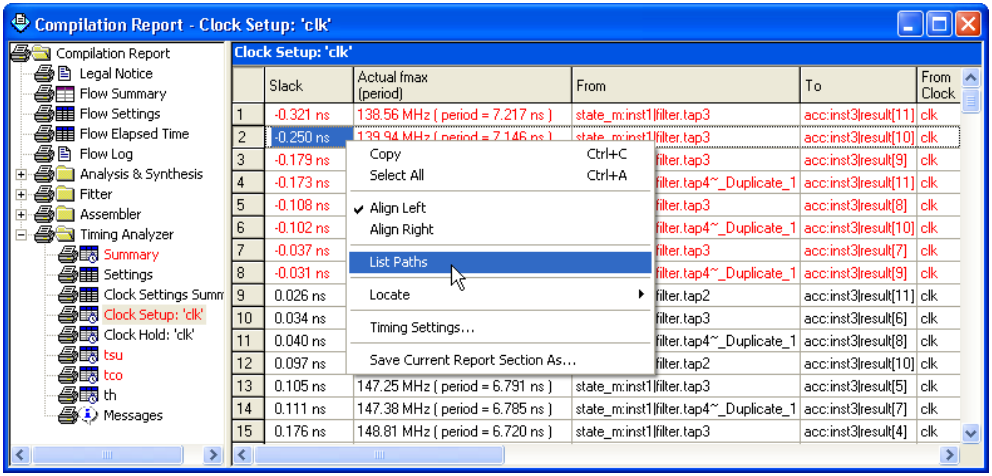
Register-to-Register Timing

If you are using TimeQuest for timing analysis, analyze the path between any two registers by using the appropriate constraints. Use the `report_timing` command to generate the required timing reports for any register-to-register path. Your design meets timing requirements when you do not have negative slack on any register-to-register path on any of the clock domains.

Timing Analysis with the Classic Timing Analyzer

If you are using the Quartus II Classic Timing Analyzer, in the Compilation Report window, use the **Timing Analyzer** section to determine whether register-to-register timing requirements are met. The **Clock Setup** folder displays you figures for the actual register-to-register for each clock as reported by the Quartus II software, and the slack delays. The paths that do not meet timing requirements are shown with a negative slack and appear in red (Figure 8–4).

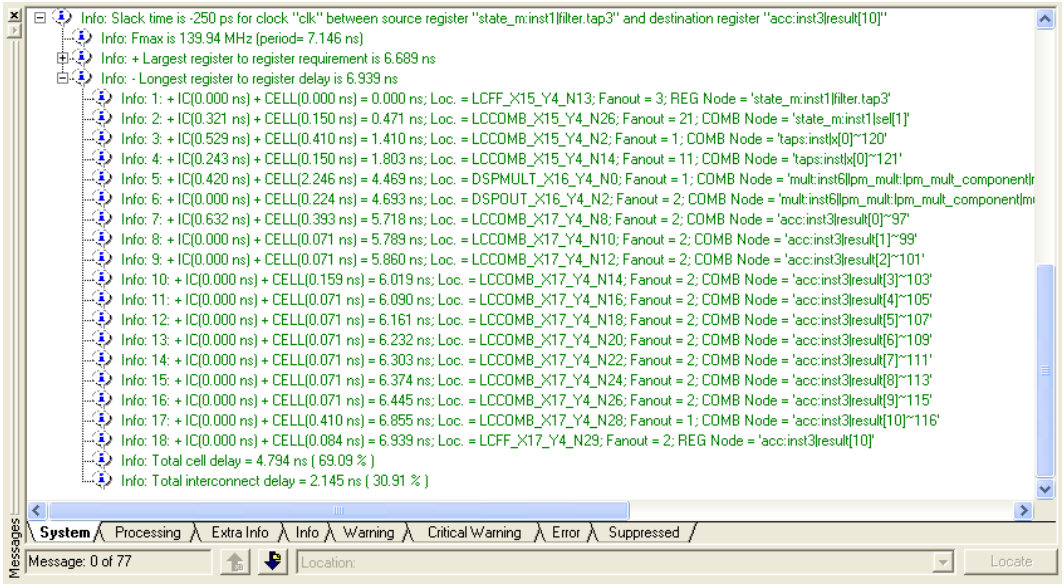
Figure 8–4. f_{MAX} Timing Analysis Report



To analyze why your timing requirements were not met, right-click on an entry in the report and click **List Paths** (Figure 8–4). A message listing the paths appears in the **System** tab of the Messages window. To expand a selection. The expanded report for the path appears (Figure 8–5). Click the ⊞ icon at the beginning of the line. Use this method to determine where the greatest delay is located along the path.

The List Paths report shows the slack time and how that slack time was calculated. By expanding the various entries, you can see the incremental delay through each node in the path as well as the total delay. The incremental delay is the sum of the interconnect delay (`IC`) and the cell delay (`CELL`) through the logic.

Figure 8–5. f_{MAX} Slack Report



To visually analyze register-to-register timing paths, right-click on a path, point to Locate, and click **Locate in Chip Planner**. For older devices, such as APEX, ACEX, FLEX 10K and MAX 7000 devices, click **Locate in Timing Closure Floorplan** to perform this analysis. The Chip Planner or Timing Closure Floorplan appears with the path highlighted. Use the **Critical Path Settings** to select which failing paths to show. To turn critical paths on or off, use the **Show Critical Paths** command.

For more information about how timing analysis results are calculated, refer to the *Quartus II Classic Timing Analyzer* or the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

You also can see the logic in a particular path by cross-probing to the RTL Viewer or Technology Map Viewer. These viewers allow you to see a gate-level or technology-mapped representation of your design netlist. To locate a timing path in one of the viewers, right-click on a path in the report, point to Locate, and click **Locate in RTL Viewer** or **Locate in Technology Map Viewer**. When you locate a timing path in the Technology Map Viewer, the annotated schematic displays the same delay information that is shown when you use the **List Paths** command.



For more information about the netlist viewers, refer to the *Analyzing Designs with Quartus II Netlist Viewers* chapter in volume 1 of the *Quartus II Handbook*.

Tips for Analyzing Failing Paths

When you are analyzing clock path failures, focus on improving the paths that show the worst slack. The Fitter works hardest on paths with the least slack. If you fix these paths, the Fitter may be able to improve the other failing timing paths in the design.

Check for particular nodes that appear in many failing paths. Look for paths that have common source registers, destination registers, or common intermediate combinational nodes. In some cases, the registers may not be identical, but are part of the same bus. In the timing analysis report panels, clicking on the **From** or **To** column headings can be helpful to sort the paths by the source or destination registers. Clicking first on **From**, then on **To**, uses the **To** register as the primary sort and **From** as the secondary sort. If you see common nodes, these nodes indicate areas of your design that might be improved through source code changes or Quartus II optimization settings. Constraining the placement for just one of the paths might decrease the timing performance for other paths by moving the common node further away in the device.

Tips for Analyzing Failing Clock Paths that Cross Clock Domains

When analyzing clock path failures, check whether these paths cross between two clock domains. This is the case if the **From Clock** and **To Clock** in the timing analysis report are different. There can also be paths that involve a different clock in the middle of the path, even if the source and destination register clock are the same. To analyze these paths in more detail, right-click on the entry in the report and click **List Paths**.

Expand the **List Paths** entry in the Messages window and analyze the largest register-to-register requirement. Evaluate the setup relationship between the source and destination (launch edge and latch edge) to determine if that is reducing the available setup time. For example, the path may go from a rising edge to a falling edge, which reduces the setup relationship by one half clock cycle.

Check if the PLL phase shift is reducing the setup requirement. You may be able to adjust this using PLL parameters and settings.

Check if the PLL compensation delay is reducing the setup relationship. If you are using the Quartus II Classic Timing Analyzer, you can direct the software to analyze this delay as clock skew by enabling Clock Latency analysis. On the **Assignments** menu, click **Settings** and choose **Timing Requirements & Options**. Click **More Settings** and turn on **Enable Clock Latency**. Typically, you must enable this option if your design results in timing violations for paths that pass between PLL clock domains. The Quartus II TimeQuest Timing Analyzer performs this analysis by default.

Paths that cross clock domains are generally protected with synchronization logic (for example, FIFOs or double-data synchronization registers) to allow asynchronous interaction between the two clock domains. In such cases, you can ignore the timing paths between registers in the two clock domains while running timing analysis, even if the clocks are related.

The Fitter attempts to optimize all failing timing paths. If there are paths that can be ignored for optimization and timing analysis, but the paths do not have constraints that instruct the Fitter to ignore them, then the Fitter tries to optimize those paths as well. In some cases, optimizing unnecessary paths can prevent the Fitter from meeting the timing requirements on timing paths that are critical to the design. It is beneficial to specify all paths that can be ignored, so the Fitter can put more effort into the paths that must meet their timing requirements instead of optimizing paths that may be ignored.



For more details about how to ignore timing paths that cross clock domains, refer to the *Quartus II Classic Timing Analyzer* chapter or the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the Quartus II handbook.

Evaluate the clock skew between the source clock and the destination clock to determine if that is reducing the available setup time. You can check the shortest and longest clock path reports to see what is causing the clock skew. Avoid using combinational logic in clock paths because it contributes to clock skew. Differences in the logic or in its routing between the source and destination can cause clock skew problems and result in warnings during compilation.

Global Routing Resources

Global routing resources are designed to distribute high-fan-out, low-skew signals (such as clocks) without consuming regular routing resources. Depending on the device, these resources may span the entire chip, or some smaller portion, such as a quadrant. The Quartus II

software attempts to assign signals to global routing resources automatically, but you may be able to make more suitable assignments manually.

Refer to the relevant device handbook for details about the number and types of global routing resources available.

Check the global signal utilization in your design to ensure that appropriate signals have been placed on global routing resources. In the Compilation Report, open the Fitter report and click the **Resource Section**. Analyze the **Global & Other Fast Signals** and **Non-Global High Fan-out Signals** reports to determine whether any changes are required.

You may be able to reduce clock skew for high fan-out signals by placing them on global routing resources. Conversely, you can reduce the insertion delay of low fan-out signals by removing them from global routing resources. Doing so can improve clock enable timing and control signal recovery/removal timing, but increases clock skew. You can also use the **Global Signal** setting in the **Assignment Editor** to control global routing resources.

Compilation Time

In long compilations, most of the time is spent in the Analysis & Synthesis and Fitter modules. Analysis & Synthesis includes synthesis netlist optimizations, if you have turned on those options. The Fitter includes two steps, placement and routing, and also includes physical synthesis if you turned on that option. The **Flow Elapsed Time** section of the Compilation Report shows how much time is spent running the Analysis & Synthesis and Fitter modules. The **Fitter Messages** report in the **Fitter** section of the Compilation Report shows specifically how much time was spent in placement and how much time was spent in routing.

The applicable messages are indicated as shown in the following example, with each time component in two-digit format:

```
Info: Fitter placement operations ending:
elapsed time = <days:hours:mins:secs>
Info: Fitter routing operations ending: elapsed
time = <days:hours:mins:secs>
```

Note that "days" are not shown if the time is less than one day.

While the Fitter is running (including Placement and Routing), hourly info messages similar to the one below are displayed every hour to indicate Fitter operations are progressing normally.

```
Info: Placement optimizations have been running for x
hour(s)
```

In this case, *x* indicates the number of hours the process has been running.

Placement is the process of finding optimum locations for the logic in your design. Routing is the process of connecting the nets between the logic in your design. There are many possible placements for the logic in a design, and finding better placements typically takes more compilation time. Good logic placement allows you to more easily meet your timing requirements and makes the design easier to route.

The elapsed time summary in the flow report indicates the percentage of time spent in each executable using parallel compilation; the metric is the average number of processors used. For example, if 50% of a compilation (by runtime) used four processors and the other 50% used one processor, this value would be $0.5 \times 4 + 0.5 \times 1 = 2.5$.

Resource Utilization Optimization Techniques (LUT-Based Devices)

After design analysis, the next stage of design optimization is to improve resource utilization. Complete this stage before proceeding to I/O timing optimization or register-to-register timing optimization. Ensure that you have already set the basic constraints described in "Initial Compilation: Required Settings" on page 8–3 before proceeding with the resource utilization optimizations discussed in this section. If a design does not fit into a specified device, use the techniques in this section to achieve a successful fit. After you optimize resource utilization and your design fits in the desired target device, optimize I/O timing as described in "I/O Timing Optimization Techniques (LUT-Based Devices)" on page 8–97. These tips are valid for all FPGA families and the MAX II family of CPLDs.

Using the Resource Optimization Advisor

The Resource Optimization Advisor provides guidance in determining settings that optimize the resource usage. To run the Resource Optimization Advisor, on the **Tools** menu, point to **Advisors**, and click **Resource Optimization Advisor**.

The Resource Optimization Advisor provides step-by-step advice about how to optimize the resource usage (logic element, memory block, DSP block, I/O, and routing) of your design. Some of the recommendations in these categories may contradict each other. Altera recommends evaluating the options, and choosing the settings that best suit your requirements.

Resolving Resource Utilization Issues Summary

Resource utilization issues can be divided into the following three categories:

- Issues relating to I/O pin utilization or placement, including dedicated I/O blocks such as PLLs or LVDS transceivers ("I/O Pin Utilization or Placement" on page 8–26).
- Issues relating to logic utilization or placement, including logic cells containing registers and look-up tables as well as dedicated logic such as memory blocks and DSP blocks ("Logic Utilization or Placement" on page 8–27).
- Issues relating to routing ("Routing" on page 8–38).

I/O Pin Utilization or Placement

Use the suggestions in the following sections to help you resolve I/O resource problems.

Use I/O Assignment Analysis

On the Processing menu, point to **Start** and click **Start I/O Assignment Analysis** to help with pin placement. The **Start I/O Assignment Analysis** command allows you to check your I/O assignments early in the design process. You can use this command to check the legality of pin assignments before, during, or after compilation of your design. If design files are available, you can use this command to perform more thorough legality checks on your design's I/O pins and surrounding logic. These checks include proper reference voltage pin usage, valid pin location assignments, and acceptable mixed I/O standards.

Common issues with I/O placement relate to the fact that differential standards have specific pin pairings, and certain I/O standards may be supported only on certain I/O banks.

If your compilation or I/O assignment analysis results in specific errors relating to I/O pins, follow the recommendations in the error message. Right-click on the message in the Messages window and click **Help** to open the Quartus II Help topic for this message.

Modify Pin Assignments or Choose a Larger Package

If a design that has pin assignments fails to fit, compile the design without the pin assignments to determine whether a fit is possible for the design in the specified device and package. You can use this approach if a Quartus II error message indicates fitting problems due to pin assignments.

If the design fits when all pin assignments are ignored or when several pin assignments are ignored or moved, you may have to modify the pin assignments for the design or choose a larger package.

If the design fails to fit because of lack of available I/Os, a successful fit can often be obtained by using a larger device package (which can be the same device density) that has more available user I/O pins.



For more information about I/O assignment analysis, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Logic Utilization or Placement

Use the suggestions in the following subsections to help you resolve logic resource problems, including logic cells containing registers and lookup tables (LUTs) as well as dedicated logic such as memory blocks and DSP blocks.

Optimize Synthesis for Area, Not Speed

If your design fails to fit because it uses too much logic, resynthesize the design to improve the area utilization. First, ensure that you have set your device and timing constraints correctly in your synthesis tool. Particularly when the area utilization of the design is a concern, ensure that you do not over-constrain the timing requirements for the design. Synthesis tools generally try to meet the specified requirements, which can result in higher device resource usage if the constraints are too aggressive.

If resource utilization is an important concern, some synthesis tools offer an easy way to optimize for area instead of speed. If you are using Quartus II integrated synthesis, choose **Balanced** or **Area** for the **Optimization Technique**. You can also specify this logic option for specific modules in your design with the Assignment Editor in cases where you want to reduce area using the **Area** setting (potentially at the expense of register-to-register timing performance) while leaving the default **Optimization Technique** setting at **Balanced** (for the best trade-off between area and speed for certain device families) or **Speed**. You can also use the **Speed Optimization Technique for Clock Domains** logic option to specify that all combinational logic in or between the specified clock domain(s) is optimized for speed.

In some synthesis tools, not specifying an f_{MAX} requirement may result in less resource utilization.



In the Quartus II software, the **Balanced** setting typically produces utilization results that are very similar to those produced by the **Area** setting, with better performance results. The **Area** setting may give better results in some unusual cases.



For information about setting timing requirements and synthesis options in Quartus II integrated synthesis and other synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*, or your synthesis software's documentation.

The Quartus II software provides additional attributes and options that can help improve the quality of your synthesis results.

Restructure Multiplexers

Multiplexers form a large portion of the logic utilization in many FPGA designs. By optimizing your multiplexed logic, you can achieve a more efficient implementation in your Altera device.

The Quartus II software provides the **Restructure Multiplexers** logic option, which can extract and optimize buses of multiplexers during synthesis. This option is available on the **Analysis & Synthesis Settings** page of the **Settings** dialog box and is useful if your design contains buses of fragmented multiplexers. This option restructures multiplexers more efficiently for area, allowing the design to implement multiplexers with a reduced number of LEs or ALMs. Using the **Restructure Multiplexers** logic option can reduce your design's register-to-register timing performance. This option is turned on automatically when you set the **Quartus II Analysis & Synthesis Optimization Technique** option to **Area** or **Balanced**. To change the default setting, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and click the appropriate option from the **Restructure Multiplexers** list to set the option globally.



For design guidelines to achieve optimal resource utilization for multiplexer designs, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. For more information about the **Restructure Multiplexers** option in the Quartus II software, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Perform WYSIWYG Resynthesis with Balanced or Area Setting

If you use another EDA synthesis tool and want to determine if the Quartus II software can remap the circuit to use fewer LEs or ALMs, follow these steps:

1. On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and turn on **Perform WYSIWYG primitive resynthesis (using optimization techniques specified in Analysis & Synthesis settings)** on the **Synthesis Netlist Optimizations** page. Or, on the Assignments menu, click **Assignment Editor**, and apply the **Perform WYSIWYG Primitive Resynthesis** logic option to a specific module in your design.

2. On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and choose **Balanced** or **Area** under Optimization Technique. Or, on the Assignments menu, click **Assignment Editor**. Set the Optimization Technique to **Balanced** or **Area** for a specific module in your design.

3. Recompile the design.

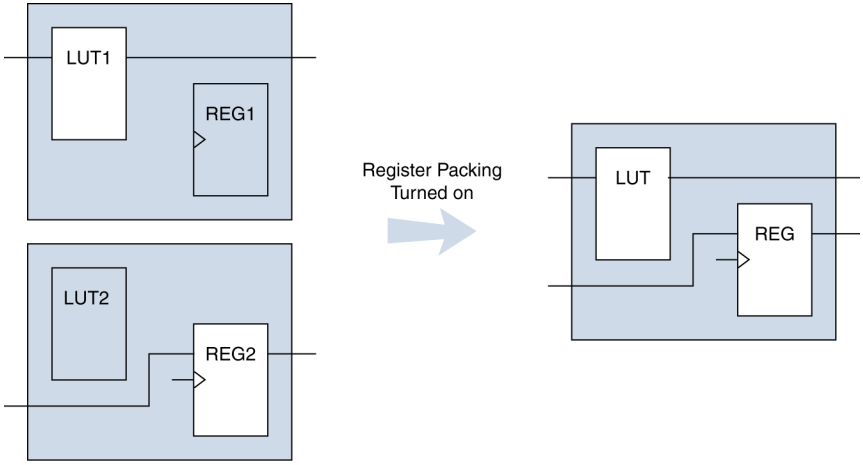


The **Balanced** setting typically produces utilization results that are very similar to the **Area** setting, with better performance results. The **Area** setting may give better results in some unusual cases. Performing WYSIWYG resynthesis for area in this way typically reduces register-to-register timing performance.

Use Register Packing

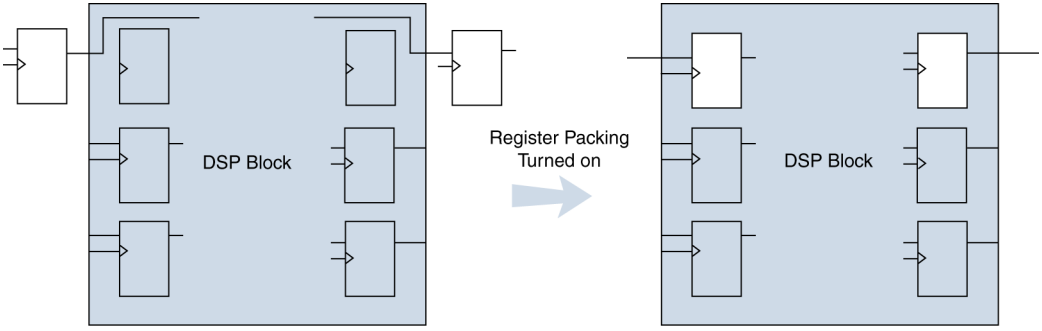
The **Auto Packed Registers** option implements the functions of two cells into one logic cell by combining the register of one cell in which only the register is used with the LUT of another cell in which only the LUT is used. Figure 8–6 shows register packing and the gain of one logic cell in the design.

Figure 8–6. Register Packing



Registers can also be packed into DSP blocks (Figure 8–7).

Figure 8–7. Register Packing in DSP Blocks



The following list shows the most common cases in which register packing helps to optimize a design:

- A LUT can be implemented in the same cell as an unrelated register with a single data input
- A LUT can be implemented in the same cell as the register that is fed by the LUT
- A LUT can be implemented in the same cell as the register that feeds the LUT
- A register can be packed into a RAM block
- A register can be packed into a DSP block
- A register can be packed into an I/O Element (IOE)

The following options are available for register packing (for certain device families):

- **Off**—Does not pack registers.
- **Normal**—Packs registers when this is not expected to hurt timing results.
- **Minimize Area**—Aggressively packs registers to reduce area.
- **Minimize Area with Chains**—Aggressively packs registers to reduce area. This option packs registers with carry chains. It also converts registers into register cascade chains and packs them with other logic to reduce area. This option is available only for Arria GX devices, the Stratix and Cyclone series of devices, and MAX II devices.
- **Auto**—This is the default setting for register packing. This setting tells the Fitter to attempt to achieve the best performance while maintaining a fit for the design in the specified device. The Fitter combines all combinational (LUT) and sequential (register) functions that benefit circuit speed. In addition, more aggressive combinations of unrelated combinational and sequential functions are performed to the extent required to reduce the area of the design to achieve a fit in the specified device. This option is available only for Arria GX devices, the Stratix and Cyclone series of devices, and MAX II devices.
- **Sparse**—In this mode, the combinational (LUT) and sequential (register) functions are combined such that the combined logic has either a combinational output or a sequential output but not both. This mode is available only for Arria GX, Stratix III, Stratix II, Cyclone III, and Cyclone II devices. This option results in a higher logic array block (LAB) usage, but might give you better timing performance because of reduced routing congestion.
- **Sparse Auto**—In this mode, the Quartus II Fitter starts with sparse mode packing, and then attempts to achieve best performance while maintaining a fit for the specified device. Later optimizations are

carried out in a way similar to the **Auto** mode. This mode is available only for Arria GX, Stratix III, Stratix II, Cyclone III, and Cyclone II devices.

Turning on register packing decreases the number of logic elements (LEs) or adaptive logic modules (ALMs) in the design, but could also decrease performance in some cases. On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**, and then click **More Settings.** Turn on **Auto Packed Registers** to turn on register packing.

The area reduction and performance results can vary greatly depending on the design. Typical results for register packing are shown in the following tables. Table 8–1 shows typical results for Arria GX devices. Table 8–2 shows typical results for Cyclone III and Cyclone II devices, and Table 8–3 shows typical results for Stratix, Stratix GX, and Cyclone devices.

The **Auto** setting performs more aggressive register packing as needed, so the typical results vary depending on the device resource utilization.

Table 8–1. Typical Register Packing Results for Arria GX, Stratix III, and Stratix II Devices

Register Packing Setting	Relative f_{MAX}	Relative ALM Count
Off	0.94	1.26
Sparse	1.01	1.04
Normal	1.00	1.00
Minimize Area	0.95	0.94
Minimize Area with Chains	0.95	0.94
Auto	1.0 until device is very full, then gradually to 0.95 as required	1.0 until device is very full, then gradually to 0.94 as required
Sparse Auto	1.01 until device is very full, then gradually to 0.95 as required	1.04 until device is very full, then gradually to 0.94 as required

Table 8–2 shows typical results for Cyclone III and Cyclone II devices.

Table 8–2. Typical Register Packing Results for Cyclone III and Cyclone II Devices (Part 1 of 2)

Register Packing Setting	Relative f_{MAX}	Relative LE Count
Off	0.97	1.40
Sparse	1.02	1.06
Normal	1.00	1.00
Minimize Area	0.96	0.93

Table 8–2. Typical Register Packing Results for Cyclone III and Cyclone II Devices (Part 2 of 2)

Register Packing Setting	Relative f_{MAX}	Relative LE Count
Minimize Area with Chains	0.94	0.91
Auto	1.0 until device is very full, then gradually to 0.94 as required	1.0 until device is very full, then gradually to 0.91 as required
Sparse Auto	1.02 until device is very full, then gradually to 0.94 as required	1.06 until device is very full, then gradually to 0.91 as required

Table 8–3 shows results for Stratix, Stratix GX, and Cyclone devices.

Table 8–3. Typical Register Packing Results for Stratix, Stratix GX and Cyclone Devices

Register Packing Setting	Relative f_{MAX}	Relative LE Count
Off	1.00	1.12
Normal	1.00	1.00
Minimize Area	0.97	0.93
Minimize Area with Chains	0.94	0.90
Auto (default)	1.0 until device is very full, then gradually to 0.94 as required	1.0 until device is very full, then gradually to 0.90 as required

Remove Fitter Constraints

A design with conflicting constraints or constraints that are difficult to meet may not fit in the targeted device. This can occur when the location or LogicLock assignments are too strict and not enough routing resources are available on the device.

In this case, use the **Routing Congestion** view in the Chip Planner or Timing Closure Floorplan (for older devices) to locate routing problems in the floorplan, then remove any location or LogicLock region assignments in that area. If your design still does not fit, the design is over-constrained. To correct the problem, remove all location and LogicLock assignments and run successive compilations, incrementally constraining the design before each compilation. You can delete specific location assignments in the Assignment Editor or the Chip Planner or the Timing Closure Floorplan. To remove LogicLock assignments in the Chip Planner (or in the Timing Closure Floorplan), in the **LogicLock Regions Window**, or, on the Assignments menu, click **Remove Assignments.** Turn on the assignment categories you want to remove from the design in the **Available assignment categories** list.



For more information about the **Routing Congestion** view in the Chip Planner and the Timing Closure Floorplan, refer to *Analyzing and Optimizing the Design Floorplan* in volume 2 of the *Quartus II Handbook*. Also refer to the Quartus II Help.

Change State Machine Encoding

State machines can be encoded using various techniques. Using binary or gray code encoding typically results in fewer state registers than one-hot encoding, which requires one register for every state bit. If your design contains state machines, changing the state machine encoding to one that uses the minimal number of registers may reduce resource utilization. The effect of state machine encoding varies depending on the way your design is structured.

If your design does not manually encode the state bits, you can specify the state machine encoding in your synthesis tool. When using Quartus II Integrated Synthesis, go to the Assignments menu and click **Settings**. In the **Category** list, select A**nalysis & Synthesis Settings** and turn on **Minimal Bits for State Machine Processing**. You also can specify this logic option for specific modules or state machines in your design with the Assignment Editor.

You can also use the following Tcl command in scripts to modify the state machine encoding.

```
set_global_assignment -name <state_machine_processing> <value>
```

In this case, <*value*> could be `AUTO`, `ONE-HOT`, `MINIMAL BITS`, or `USER-ENCODE`.

Flatten the Hierarchy During Synthesis

Synthesis tools typically provide the option of preserving hierarchical boundaries, which may be useful for verification or other purposes. However, optimizing across hierarchical boundaries allows the synthesis tool to perform the most logic minimization, which can reduce area. Therefore, to achieve the best results, flatten your design hierarchy whenever possible. If you are using Quartus II integrated synthesis, ensure that the **Preserve Hierarchical Boundary** logic option is turned off, that is, make sure that you have not turned on the option in the Assignment Editor or with Tcl assignments. If you are using Quartus II incremental compilation, you cannot flatten your design across design partitions. Incremental compilation always preserves the hierarchical boundaries between design partitions. Follow Altera's recommendations for design partitioning, such as registering partition boundaries to reduce the effect of cross-boundary optimizations.



For more information about using incremental compilation and recommendations for design partitioning, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. If you are using an incremental synthesis flow that requires separate hierarchy blocks, you can find additional recommendations for design partitioning in the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook.*

Retarget Memory Blocks

If the design fails to fit because it runs out of device memory resources, your design may require a certain type of memory the device does not have. For example, a design that requires two M-RAM blocks can be targeted to a Stratix EP1S10 device, which has only one M-RAM block. By building one of the memories with a different size memory block, such as an M4K memory block, you might obtain a fit.

If the memory block was created with the MegaWizard® Plug-In Manager, open the MegaWizard Plug-In Manager and edit the RAM block type so it targets a new memory block size.

ROM and RAM memory blocks can also be inferred from your HDL code, and your synthesis software can place large shift registers into memory blocks by inferring the `altshift_taps` megafunction. This inference can be turned off in your synthesis tool to cause the memory to be placed in logic instead of in memory blocks. To disable inference when using Quartus II integrated synthesis, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis**, and turn off the **Auto RAM Replacement**, **Auto ROM Replacement**, or **Auto Shift Register Replacement** logic option as appropriate for your project. Or, disable the option for a specific entity in the Assignment Editor.

Depending on your synthesis tool, you can also set the RAM block type for inferred memory blocks. In Quartus II integrated synthesis, set the **ramstyle** attribute to the desired memory type for the inferred RAM blocks, or set the option to **logic** to implement the memory block in standard logic instead of a memory block.

Consider the resource utilization by hierarchy in the report file, and determine whether there is an unusually high register count in any of the modules. Some coding styles may prevent the Quartus II software from inferring RAM blocks from the source code because of their architectural implementation, and forcing the software to implement the logic in flipflops, instead. As an example, a function such as an asynchronous reset on a register bank might make it incompatible with the RAM blocks

in the device architecture, so that the register bank is implemented in flipflops. Often it is possible to move a large register bank into RAM by slight modification of associated logic.



For more information about memory inference control in other synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook,* or your synthesis software's documentation. For more information about coding styles and HDL examples that ensure memory inference, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Use Physical Synthesis Options to Reduce Area

The physical synthesis options for fitting may help you decrease the resource usage; additional optimizations for fitting are available. When you enable these settings for physical synthesis for fitting, the Quartus II software makes placement-specific changes to the netlist that reduce resource utilization for a specific Altera device.

The following physical synthesis optimizations for fitting are available:

- Physical synthesis for combinational logic
- Map logic into memory

On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**, and specify the physical synthesis optimization options on the **Physical Synthesis Optimizations** page. You can also specify the physical synthesis effort, which sets the level of physical synthesis optimization that you want the Quartus II software to perform.

The Perform physical synthesis for combinational logic for fitting option allows the Quartus II Fitter to resynthesize the combinational logic in a design to reduce the resource utilization to help achieve a fit.

The Map logic into memory for fitting option allows the Quartus II Fitter to automatically map logic into unused memory blocks during fitting, reducing the number of logic elements required to implement the design.

Retarget or Balance DSP Blocks

A design may not fit because it requires too many DSP blocks. All DSP block functions can be implemented with logic cells, so you can retarget some of the DSP blocks to logic to obtain a fit.

If the DSP function was created with the MegaWizard Plug-In Manager, open the MegaWizard Plug-In Manager and edit the function so it targets logic cells instead of DSP blocks. The Quartus II software uses the `DEDICATED_MULTIPLIER_CIRCUITRY` megafunction parameter to control the implementation.

DSP blocks also can be inferred from your HDL code for multipliers, multiply-adders, and multiply-accumulators. This inference can be turned off in your synthesis tool. When you are using Quartus II integrated synthesis, you can disable inference by turning off the **Auto DSP Block Replacement** logic option for your whole project. On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and turn off **Auto DSP Block Replacement**. Alternatively, you can disable the option for a specific block with the Assignment Editor.



For more information about disabling DSP block inference in other synthesis tools, refer to the appropriate chapter in the Synthesis section in volume 1 of the *Quartus II Handbook*, or your synthesis software's documentation.

The Quartus II software also offers the **DSP Block Balancing** logic option, which implements DSP block elements in logic cells or in different DSP block modes. The default **Auto** setting allows DSP block balancing to convert the DSP block slices automatically as appropriate to minimize the area and maximize the speed of the design. You can use other settings for a specific node or entity, or on a project-wide basis, to control how the Quartus II software converts DSP functions into logic cells and DSP blocks. Using any value other than **Auto** or **Off** overrides the `DEDICATED_MULTIPLIER_CIRCUITRY` parameter used in megafunction variations.



For more details about the Quartus II logic options described in this section, refer to the Quartus II Help.

Optimize Source Code

If your design does not fit because of logic utilization, and the methods described in the preceding sections do not sufficiently improve the resource utilization of the design, modify the design at the source to achieve the desired results. You can often improve logic significantly by making design-specific changes to your source code. This is typically the most effective technique for improving the quality of your results.

If your design does not fit into available LEs or ALMs, but you have unused memory or DSP blocks, check to see if you have code blocks in your design that describe memory or DSP functions that are not being

inferred and placed in dedicated logic. You may be able to modify your source code to allow these functions to be placed into dedicated memory or DSP resources in the target device.

Ensure that your state machines are recognized as state machine logic and optimized appropriately in your synthesis tool. State machines that are recognized are generally optimized better than if the synthesis tool treats them as generic logic. In the Quartus II software, you can check for the **State Machine** report under **Analysis & Synthesis** in the Compilation Report. This report provides details, including the state encoding for each state machine that was recognized during compilation. If your state machine is not being recognized, you may need to change your source code to enable it to be recognized.



For coding style guidelines including examples of HDL code for inferring memory and DSP functions, refer to the *Instantiating Altera Megafunctions* and the *Inferring Altera Megafunctions* sections of the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. For guidelines and sample HDL code for state machines, refer to the *General Coding Guidelines* section of the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Use a Larger Device

If a successful fit cannot be achieved because of a shortage of LEs or ALMs, memory, or DSP blocks, you may need to use a larger device.

Routing

Use the suggestions in the following subsections to help you resolve routing resource problems.

Set Auto Register Packing to Auto

This option is useful for reducing LE or ALM count in a design. This option is available for Arria GX devices and for the Cyclone and Stratix series of devices. On the Settings menu, select **Fitter Settings**. Click **More Settings**. From the options list, select **Auto Register Packing** and select the **Auto** option from the drop-down menu.

When you choose a register packing setting to perform more register packing than the **Auto** setting, the extra register packing may affect the routability of the design as an unintended result. The **Minimize the area with chains** setting restricts placement and reduces routability significantly more than using the **Minimize Area** setting. For more information about register packing, refer to "Use Register Packing" on page 8–29.

Set Fitter Aggressive Routability Optimizations to Always

If routing resources are resulting in no-fit errors, use this option to reduce routing wire utilization. On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**. Click **More Settings**. In the More Fitter Settings dialog box, set **Fitter Aggressive Routability Optimizations** to **Always** and click **OK**.

On average, in Arria GX and Stratix II devices, this option saves approximately 3% wire utilization but can reduce performance by approximately 1%. In Stratix III devices, this option saves approximately 6% wire utilization, at the same time reducing the performance by approximately 3%. In Cyclone III devices, using this option saves approximately 4.5% wire utilization while reducing the performance by about 4%.

These optimizations are used automatically when the Fitter performs more than one fitting attempt, but turning the option on increases the optimization effort on the first fitting attempt. This option also ensures that the Quartus II software uses maximum optimization to reduce routability, even if the Fitter Effort is set to **Auto Fit**.

You can modify the Placement Effort Multiplier using the following Tcl command; <*value*> can be any positive, non-zero number.

```
set_global_assignment \
    -name PLACEMENT_EFFORT_MULTIPLIER <value>
```

Increase Placement Effort Multiplier

Increasing the placement effort can improve the routability of the design, allowing the software to route a design that otherwise requires too many routing resources. On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**. Click **More Settings**. In the **More Fitter Settings** dialog box, increase the value of the **Placement Effort Multiplier** to increase placement effort. The default value is `1.0`. Legal values must be greater than `0` and can be non-integer values. Higher numbers increase compilation time but may improve placement quality. For example, a value of `4` increases fitting time by approximately 2 to 4 times but may increase the quality of results. Increasing the placement effort multiplier does not tend to improve timing optimization unless the design also has very high routing resource usage.

Increased effort is used automatically when the Fitter performs more than one fitting attempt. Setting a multiplier higher than one (before compilation) increases the optimization effort on the first fitting attempt.

The second and third fitting loops increase the Placement Effort Multiplier to 4 and then to 16. These loops result in increased compilation times, with possible improvement in the quality of placement.

Increase Router Effort Multiplier

The Router Effort Multiplier controls how quickly the router tries to find a valid solution. The default value is 1.0, and legal values must be greater than 0. Numbers higher than 1 (as high as 3 is generally reasonable) may improve routing quality at the expense of run-time on difficult-to-route circuits. Numbers closer to 0 (for example, 0.1) can reduce router runtime, but usually reduce routing quality slightly. Experimental evidence shows that a multiplier of 3.0 reduces overall wire usage by about 2%. There is usually no gain in performance beyond a multiplier value of 3.

You can set the Router Effort Multiplier to a value higher than the default value for difficult-to-route designs. To set the Router Effort Multiplier, from the Assignments menu, click **Settings**, and then click **Fitter Settings**. Click the **More Settings** button. From the options available, select **Router Effort Multiplier** and edit the value in the dialog box that appears.

You can modify the Router Effort Multiplier by the following Tcl command, where <*value*> can be any positive, non-zero number.

```
set_global_assignment \
    -name ROUTER_EFFORT_MULTIPLIER  <value>
```

Remove Fitter Constraints

A design with conflicting constraints or constraints that are difficult to meet may not fit the targeted device. This can occur when location or LogicLock assignments are too strict and there are not enough routing resources.

In this case, use the **Routing Congestion** view in the Chip Planner or the Timing Closure Floorplan (for older devices) to locate routing problems in the floorplan, then remove all location and LogicLock region assignments from that area. If your design still does not fit, the design is over-constrained. To correct the problem, remove all location and LogicLock assignments and run successive compilations, incrementally constraining the design before each compilation. You can delete specific location assignments in the Assignment Editor or the Chip Planner or Timing Closure Floorplan (for supported devices). Remove LogicLock assignments in the Chip Planner (or in the Timing Closure Floorplan), in

the **LogicLock Regions Window**, or, on the Assignments menu, click **Remove Assignments**. Turn on the assignment categories you want to remove from the design in the **Available assignment categories** list.



For more information about the **Routing Congestion** view in the Chip Planner or the Timing Closure Floorplan, refer to the "Routing Congestion View" section in the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*. You can also refer to the Quartus II Help.

Set Maximum Router Timing Optimization Level

To improve routability in cases in which the router did not pick up the optimal routing lines, set the Router Timing Optimization Level to **Maximum**. This setting determines how aggressively the router tries to meet timing requirements. Setting this option to Maximum can increase design speed slightly, at the cost of increased compilation time. Setting this option to **Minimum** can reduce compilation time, at the cost of slightly reduced design speed. The default value is **Normal**.

To modify the Router Timing Optimization level, on the Assignments menu, click **Settings**. The **Settings** dialog box appears. In the **Category** list, click **Fitter Settings**. Click on the **More Settings** tab. From the available settings, select **Router Timing Optimization Level** and choose the required setting from the drop-down menu.

Optimize Synthesis for Area, Not Speed

In some cases, resynthesizing the design to improve the area utilization can also improve the routability of the design. First, ensure that you have set your device and timing constraints correctly in your synthesis tool. Ensure that you do not over-constrain the timing requirements for the design, particularly when the area utilization of the design is a concern. Synthesis tools generally try to meet the specified requirements, which can result in higher device resource usage if the constraints are too aggressive.

If resource utilization is important to improving the routing results in your design, some synthesis tools offer an easy way to optimize for area instead of speed. If you are using Quartus II integrated synthesis, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and choose **Balanced** or **Area** under **Optimization Technique**.

You can also specify this logic option for specific modules in your design with the Assignment Editor in cases where you want to reduce area using the **Area** setting (potentially at the expense of register-to-register timing

performance). You can apply the setting to specific modules while leaving the default Optimization Technique setting at **Balanced** (for the best trade-off between area and speed for certain device families) or **Speed**. You can also use the **Speed Optimization Technique for Clock Domains** logic option to specify that all combinational logic in or between the specified clock domain(s) is optimized for speed.



In the Quartus II software, the **Balanced** setting typically produces utilization results that are very similar to those obtained with the **Area** setting, with better performance results. The Area setting may give better results in some unusual cases.

In some synthesis tools, not specifying an f_{MAX} requirement may result in less resource utilization which may improve routability.



For information about setting timing requirements and synthesis options in Quartus II integrated synthesis and other synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*, or your synthesis software's documentation.

Optimize Source Code

If your design does not fit because of routing problems, and the methods described in the preceding sections do not sufficiently improve the routability of the design, modify the design at the source to achieve the desired results. You can often improve results significantly by making design-specific changes to your source code, such as duplicating logic or changing the connections between blocks that require significant routing resources.

Use a Larger Device

If a successful fit cannot be achieved because of a shortage of routing resources, you may need to use a larger device.

Timing Optimization Techniques (LUT-Based Devices)

This section contains guidelines if your design does not meet its timing requirements.

Timing Optimization Advisor

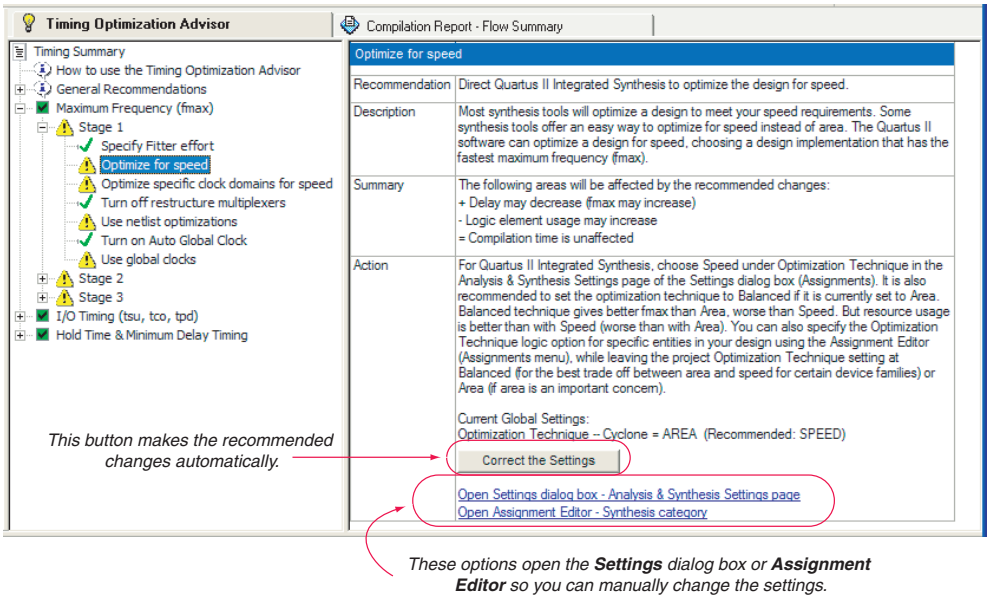
The Timing Optimization Advisor guides you to make settings that optimize your design to meet your timing requirements. To run the Timing Optimization Advisor, on the Tools menu, point to Advisors, and click on **Timing Optimization Advisor**. This advisor describes many of the suggestions made in this section.

When you open the Timing Optimization Advisors after compilation, you find recommendations to improve the timing performance of your design. Some of the recommendations in these advisors may contradict each other. Altera recommends evaluating these options and choosing the settings that best suit the given requirements.

I/O Timing Optimization

The example in Figure 8–8 shows the Timing Optimization Advisor after compiling a design that meets its frequency requirements, but requires settings changes to improve the timing.

Figure 8–8. Timing Optimization Advisor



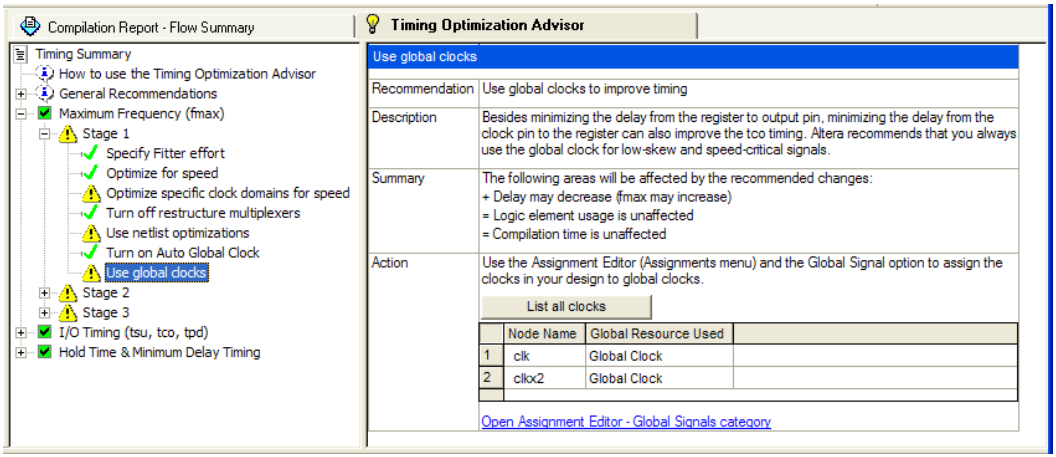
When you expand one of the categories in the Advisor, such as **Maximum Frequency (fmax)** or **I/O Timing (tsu, tco, tpd)**, the recommendations are divided into stages. The stages show the order in which you should apply the recommended settings. The first stage contains the options that are easiest to change, make the least drastic changes to your design optimization, and have the least effect on compilation time. Icons indicate whether each recommended setting has been made in the current project. In Figure 8–8, the check mark icons in the list of recommendations for **Stage 1** indicate recommendations that are already implemented. The warning icons indicate recommendations that are not followed for this compilation. The information icon indicates general suggestions. For

these entries, the advisor does not report whether these recommendations were followed, but instead explains how you can achieve better performance. Refer to the "How to use" page in the Advisor for a legend that provides more information for each icon.

There is a link from each recommendation to the appropriate location in the Quartus II user interface where you can change the settings. For example, consider the **Synthesis Netlist Optimizations** page of the **Settings** dialog box or the **Global Signals** category in the Assignment Editor. This approach provides the most control over which settings are made, and helps you learn about the settings in the software. In some cases, you can also use the **Correct the Settings** button, shown in the advisor in Figure 8–8, to automatically make the suggested change to global settings.

For some entries in the advisor, a button appears that allows you to further analyze your design and gives you more information. For example, Figure 8–9 shows the guidelines for the **Use Global Clocks** entry, after the user has clicked **List all clocks**. The advisor provides a table with the clocks in the design, and indicates whether they have been assigned a timing constraint.

Figure 8–9. Timing Optimization Advisor



The next stage of design optimization focuses on I/O timing. Ensure that you have made the appropriate assignments as described in "Initial Compilation: Required Settings" on page 8–3, and that the resource utilization is satisfactory, before proceeding with I/O timing optimization. The suggestions given in this section are applicable to all Altera FPGA families and to the MAX II family of CPLDs.

Because changes to the I/O paths affect the internal register-to-register timing, complete this stage before proceeding to the register-to-register timing optimization stage as described in the "Register-to-Register Timing Optimization Techniques (LUT-Based Devices)" on page 8–52.

The options presented in this section address how to improve I/O timing, including the setup delay (t_{SU}), hold time (t_H), and clock-to-output (t_{CO}) parameters.

Improving Setup and Clock-to-Output Times Summary

Table 8–4 shows the recommended order in which to use techniques to reduce t_{SU} and t_{CO} times. Check marks indicate which timing parameters are affected by each technique. Reducing t_{SU} times increases hold (t_H) times.

Table 8–4. Improving Setup and Clock-to-Output Times *Note (1)* **(Part 1 of 2)**

Technique	Affects t_{SU}	Affects t_{CO}
Ensure that the appropriate constraints are set for the failing I/Os (page 8–4)	✓	✓
Use timing-driven compilation for I/O (page 8–47)	✓	✓
Use fast input register (page 8–47)	✓	—
Use fast output register, fast output enable register, and fast OCT register (page 8–47)	—	✓
Decrease the value of **Input Delay from Pin to Input Register** or set **Decrease Input Delay to Input Register = ON** (page 8–49)	✓	—
Decrease the value of **Input Delay from Pin to Internal Cells**, or set **Decrease Input Delay to Internal Cells = ON** (page 8–49)	✓	—
Decrease the value of **Delay from Output Register to Output Pin**, or set **Increase Delay to Output Pin = OFF** (page 8–49)	—	✓
Increase the value of **Input Delay from Dual-Purpose Clock Pin to Fan-Out Destinations** (page 8–51)	✓	—
Use PLLs to shift clock edges (page 8–51)	✓	✓
Use the Fast Regional Clock option (page 8–52)	—	✓

Table 8–4. Improving Setup and Clock-to-Output Times *Note (1)* *(Part 2 of 2)*		
Technique	**Affects t_{SU}**	**Affects t_{CO}**
For MAX II devices, set **Guarantee I/O paths** to zero, **Hold Time at Fast Timing Corner** to OFF, or When t_{SU} and t_{PD} constraints permit (page 8–52)	✓	—
Increase the value of **Delay to output enable pin** or set **Increase delay to output enable pin** (page 8–51)	—	✓

Note to Table 8–4:
(1) These options may not apply to all device families.

Timing-Driven Compilation

To perform IOC timing optimization using the **Optimize IOC Register Placement For Timing** option, perform the following steps.

1. On the Assignments menu, click **Settings**.
2. In the Category list, select **Fitter Settings** and click **More Settings**.
3. In the **More Fitter Settings** dialog box, under **Existing options settings**, select **Optimize IOC Register Placement for Timing**.

This option moves registers into I/O elements if required to meet t_{SU} or t_{CO} assignments, duplicating the register if necessary (as in the case where a register fans out to multiple output locations). This option is enabled by default and is a global setting. The option does not apply to MAX II devices because they do not contain I/O registers.

For APEX™ 20KE and APEX 20KC devices, if the I/O register is not available, the Fitter tries to move the register into the logic array block (LAB) adjacent to the I/O element.

The **Optimize IOC Register Placement for Timing** option affects only pins that have a t_{SU} or t_{CO} requirement. Using the I/O register is possible only if the register directly feeds a pin or is fed directly by a pin. This setting does not affect registers with any of the following characteristics:

- Have combinational logic between the register and the pin
- Are part of a carry or cascade chain
- Have an overriding location assignment
- Use the synchronous load or asynchronous clear ports of APEX 20K and APEX II devices

- Are input registers that use the synchronous load port and the value is not 1 (in device families where the port is available, other than APEX 20K, APEX II, and FLEX® 6000 devices)
- Use the asynchronous load port and the value is not 1 (in device families where the port is available)

Registers with the characteristics listed are optimized using the regular Quartus II Fitter optimizations.

Fast Input, Output and Output Enable Registers

You can place individual registers in I/O cells manually by making fast I/O assignments with the Assignment Editor. For an input register, use the **Fast Input Register** option; for an output register, use the **Fast Output Register** option; and for an output enable register, use the **Fast Output Enable Register** option. Stratix II devices also support the Fast OCT (on-chip termination) Register option. In MAX II devices, which have no I/O registers, these assignments lock the register into the LAB adjacent to the I/O pin if there is a pin location assignment for that I/O pin.

If the fast I/O setting is on, the register is always placed in the I/O element. If the fast I/O setting is off, the register is never placed in the I/O element. This is true even if the **Optimize IOC Register Placement for Timing** option is turned on. If there is no fast I/O assignment, the Quartus II software determines whether to place registers in I/O elements if the **Optimize IOC Register Placement for Timing** option is turned on.

The four fast I/O options (**Fast Input Register**, **Fast Output Register**, **Fast Output Enable Register**, and **Fast OCT Register**) also can be used to override the location of a register that is in a LogicLock region, and force it into an I/O cell. If this assignment is applied to a register that feeds multiple pins, the register is duplicated and placed in all relevant I/O elements. In MAX II devices, the register is duplicated and placed in each distinct LAB location that is next to an I/O pin with a pin location assignment.

Programmable Delays

Various programmable delay options can be used to minimize the t_{SU} and t_{CO} times. For Arria GX devices, Stratix and Cyclone series devices, and MAX II devices, the Quartus II software automatically adjusts the applicable programmable delays to help meet timing requirements. For APEX series devices, the default values are set to avoid any hold time problems. Programmable delays are advanced options that you should

use only after you compile a project, check the I/O timing, and determine that the timing is unsatisfactory. For detailed information about the effect of these options, refer to the device family handbook or data sheet.

After you have made a programmable delay assignment and compiled the design, you can view the value of every delay chain for every I/O pin in the **Delay Chain Summary** section of the Compilation Report.

You can assign programmable delay options to supported nodes with the Assignment Editor. You also can view and modify the delay chain setting for the target device with the Chip Planner and Resource Property Editor. When you use the Resource Property Editor to make changes after performing a full compilation, recompiling the entire design is not necessary; you can save changes directly to the netlist. Because these changes are made directly to the netlist, the changes are not made again automatically when you recompile the design. The change management features allow you to reapply the changes on subsequent compilations.

Though the programmable delays in Stratix III devices are user-controllable, Altera recommends their use for advanced users only. However, the Quartus II software may use the programmable delays internally during the Fitter phase.



For more details about Stratix III programmable delays, refer to the *Stratix III Device Handbook* and *AN 474: Stratix III Programmable I/O Delay Settings in Quartus II*.



For more information about using the Chip Planner and Resource Property Editor, refer to the *Design Analysis and Engineering Change Management with Chip Planner* chapter in volume 3 of the *Quartus II Handbook*.

Table 8–5 summarizes the programmable delays available for Altera devices.

Table 8–5. Programmable Delays for Altera Devices (Part 1 of 2)

Programmable Delay	Description	I/O Timing Impact	Devices
Decrease input delay to input register	Decreases propagation delay from an input pin to the data input of the input register in the I/O cell associated with the pin. Applied to an input/bidirectional pin or register it feeds.	Decreases t_{SU} Increases t_H	• Stratix • Stratix GX • Cyclone • MAX 7000B • APEX II • APEX 20KE • APEX 20KC • Mercury™
Input delay from pin to input register	Sets propagation delay from an input pin to the data input of the input register implemented in the I/O cell associated with the pin. Applied to an input/bidirectional pin.	Changes t_{SU} Changes t_H	• Arria GX • Stratix II • Stratix II GX • Cyclone III • Cyclone II
Decrease input delay to internal cells	Decreases the propagation delay from an input or bidirectional pin to logic cells and embedded cells in the device. Applied to an input/bidirectional pin or register it feeds.	Decreases t_{SU} Increases t_H	• Stratix • Stratix GX • Cyclone • APEX II • APEX 20KE • APEX 20KC • ACEX® 1K • FLEX 10K® • FLEX 6000 • Mercury
Input delay from pin to internal cells	Sets the propagation delay from an input or bidirectional pin to logic and embedded cells in the device. Applied to an input or bidirectional pin.	Changes t_{SU} Changes t_H	• Stratix II • Stratix II GX • Cyclone III • Cyclone II • MAX II
Decrease input delay to output register	Decreases the propagation delay from the interior of the device to an output register in an I/O cell. Applied to an input/bidirectional pin or register it feeds.	Decreases t_{PD}	• Arria GX • Stratix • Stratix GX • APEX II • APEX 20KE • APEX 20KC
Increase delay to output enable pin	Increases the propagation delay through the tri-state output to the pin. The signal can either come from internal logic or the output enable register in an I/O cell. Applied to an output/bidirectional pin or register feeding it.	Increases t_{CO}	• Stratix • Stratix GX • APEX II • Mercury

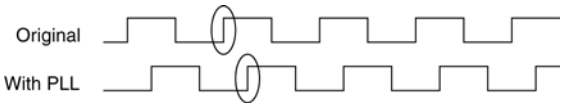
Table 8–5. Programmable Delays for Altera Devices (Part 2 of 2)

Programmable Delay	Description	I/O Timing Impact	Devices
Delay to output enable pin	Sets the propagation delay to an output enable pin from internal logic or the output enable register implemented in an I/O cell.	Changes t_{CO}	• Arria GX • Stratix II • Stratix II GX • Cyclone III
Increase delay to output pin	Increases the propagation delay to the output or bidirectional pin from internal logic or the output register in an I/O cell. Applied to output/bidirectional pin or register feeding it.	Increases t_{CO}	• Stratix • Stratix GX • Cyclone • APEX II • APEX 20KE • APEX 20KC • Mercury
Delay from output register to output pin	Sets the propagation delay to the output or bidirectional pin from the output register implemented in an I/O cell. This option is off by default.	Changes t_{CO}	• Arria GX • Stratix II • Stratix II GX • Cyclone III • Cyclone II
Increase input clock enable delay	Increases the propagation delay from the interior of the device to the clock enable input of an I/O input register.	N/A	• Stratix • Stratix GX • APEX II • APEX 20KE • APEX 20KC
Input delay from dual purpose clock pin to fan-out destinations	Sets the propagation delay from a dual-purpose clock pin to its fan-out destinations that are routed on the global clock network. Applied to an input or bidirectional dual-purpose clock pin.	N/A	• Cyclone III • Cyclone II
Increase output clock enable delay	Increases the propagation delay from the interior of the device to the clock enable input of the I/O output register and output enable register.	N/A	• Stratix • Stratix GX • APEX II • APEX 20KE • APEX 20KC
Increase output enable clock enable delay	Increases the propagation delay from the interior of the device to the clock enable input of an output enable register.	N/A	• Stratix • Stratix GX
Increase t_{ZX} delay to output pin	Used for zero bus-turnaround (ZBT) by increasing the propagation delay of the falling edge of the output enable signal.	Increases t_{CO}	• Stratix • Stratix GX • APEX II • Mercury

Use PLLs to Shift Clock Edges

Using a PLL typically improves I/O timing automatically. If the timing requirements are still not met, most devices allow the PLL output to be phase shifted in order to change the I/O timing. Shifting the clock backwards gives a better t_{CO} at the expense of t_{SU}, while shifting it forward gives a better t_{SU} at the expense of t_{CO} and t_{H}. Refer to Figure 8–10. This technique can be used only in devices that offer PLLs with the phase shift option.

Figure 8–10. Shift Clock Edges Forward to Improve t_{SU} at the Expense of t_{CO}



You can achieve the same type of effect in certain devices using the programmable delay called **Input Delay from Dual Purpose Clock Pin to Fan-Out Destinations**, described in Table 8–5.

Use Fast Regional Clock Networks and Regional Clocks Networks

Stratix EP1S25, EP1S20, and EP1S10 devices, and Stratix GX EP1SGX25 and EP1SGX10 devices, contain two fast regional clock networks, `FCLK[1..0]`, in each quadrant, fed by input pins that can connect to other fast regional clock networks.

In Stratix EP1S30, Stratix GX EP1SGX40, and larger devices in both families, there are two fast regional clock networks in each half-quadrant. Dedicated FCLK input pins feed these clock nets directly. Stratix fast regional clocks have less delay to I/O elements than do regional and global clocks, and are used for high fan-out control signals.

Stratix III, Stratix II GX, and Stratix II devices provide 32 regional clock networks. Eight regional clock networks, `RCLK[7..0]`, in each quadrant of the device are driven by the dedicated input pins `CLK[15..0]`, by PLL outputs, or by internal logic. These regional clock networks provide the lowest clock delay and skew for logic contained in a single quadrant. Placing clocks on these low-skew and low-delay clock nets provides better t_{CO} performance.

Change How Hold Times are Optimized for MAX II Devices

For MAX II devices, you can use the **Guarantee I/O paths have zero hold time at Fast Timing Corner** option to control how hold time is optimized by the Quartus II software. On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**. Click **More Settings.** In the More Fitter Settings dialog box, set the option globally. Or, on the Assignments menu, click **Assignment Editor** to set this option for specific I/Os.

The option controls whether the Fitter uses timing-driven compilation to optimize a design to achieve a zero hold time for I/Os that feed globally clocked registers at the fast (best-case) timing corner, even in the absence of any user timing assignments. When this option is set to **On** (default), the Fitter guarantees zero hold time (t_H) for I/Os feeding globally clocked registers at the fast timing corner, at the expense of possibly violating t_{SU} or t_{PD} timing constraints. When this option is set to **When tsu and tpd constraints permit**, the Fitter achieves zero hold time for I/Os feeding globally clocked registers at the fast timing corner only when t_{SU} or t_{PD} timing constraints are not violated. When this option is set to **Off**, designs are optimized to meet user timing assignments only.

By setting this option to **Off** or **When tsu and tpd constraints permit**, you improve t_{SU} at the expense of t_H.

Register-to-Register Timing Optimization Techniques (LUT-Based Devices)

The next stage of design optimization is to improve register-to-register (f_{MAX}) timing. There are a number of options available if the performance requirements are not achieved after compilation.



When using the Quartus II TimeQuest Timing Analyzer, register-to-register timing optimization is the same as maximizing the slack on the clock domains in your domain. You can use the techniques described in this section to improve the slack on different timing paths in your design.

Before optimizing your design, you should understand the structure of your design as well as the type of logic affected by each optimization. An optimization can decrease performance if the optimization does not benefit your logic structure.

Improving Register-to-Register Timing Summary

The choice of options and settings to improve the timing margin (slack) or to improve register-to-register timing depends on the failing paths in the design. To achieve the results that best approximate your performance requirements, apply the following techniques, and compile the design after each:

1. Ensure that your timing assignments are complete. For details, refer to "Timing Requirement Settings" on page 8–4.
2. Ensure that you have reviewed all warning messages from your initial compilation, and have checked for ignored timing assignments. Refer to "Design Analysis" on page 8–14 for details and fix any of these problems before proceeding with optimization.
3. Apply netlist synthesis optimization options and physical synthesis (page 8–54).
4. Try multiple different Fitter seeds (page 8–62). You can omit this step if a large number of critical paths are failing, or if paths are failing badly.
5. Apply the following synthesis options to optimize for speed:
   - Optimize Synthesis for Speed, Not Area (page 8–58)
   - Flatten the Hierarchy During Synthesis (page 8–59)
   - Set the Synthesis Effort to High (page 8–60)
   - Change State Machine Encoding (page 8–60)
   - Duplicate Logic for Fan-Out Control (page 8–60)
   - Prevent Shift Register Inference (page 8–61)
   - Use Other Synthesis Options Available in Your Synthesis Tool (page 8–61)
6. Make LogicLock assignments (page 8–64) to control placement.
7. Make design source code modifications to fix areas of the design that are still failing timing requirements by significant amounts (page 8–63).
8. Make location assignments, or, as a last resort, perform manual placement by back-annotating the design (page 8–67).



You can use the Design Space Explorer (DSE) to automate the process of running several different compilations with different settings. For more information, refer to the Design Space Explorer chapter in volume 2 of the *Quartus II Handbook*.

If these techniques do not achieve performance requirements, additional design source code modifications may be required (page 8–63).

Synthesis Netlist Optimizations and Physical Synthesis Optimizations

The Quartus II software offers advanced netlist optimization options, including physical synthesis. Various netlist optimizations can help improve the performance of many designs regardless of the synthesis tool used. Netlist optimizations can be applied both during synthesis and during fitting.

The synthesis netlist optimizations occur during the synthesis stage of the Quartus II compilation. Operating either on the output from another EDA synthesis tool or as an intermediate step in Quartus II integrated synthesis, these optimizations make changes to the synthesis netlist to improve either area or speed, depending on your selected optimization technique.

The following synthesis netlist optimizations are available during the synthesis stage:

- WYSIWYG primitive resynthesis (for netlists from third-party EDA synthesis tools)
- Gate-level register retiming

On the Assignments menu, click **Settings**. In the **Category** list, expand **Analysis & Synthesis Settings** and select **Synthesis Netlist Optimizations** to view and modify the synthesis netlist optimization options.

If you use another EDA synthesis tool and want to determine if the Quartus II software can remap the circuit to improve performance, you can use the **Perform WYSIWYG Primitive Resynthesis** option. This option directs the Quartus II software to unmap the LEs in an atom netlist to logic gates, and then map the gates back to Altera-specific primitives. Using Altera-specific primitives enables the Fitter to remap the circuits using architecture-specific techniques.

To turn on the **Perform WYSIWYG Primitive Resynthesis** option, on the Assignments menu, click **Settings**. In the **Category** list, expand **Analysis & Synthesis Settings** and select **Synthesis Netlist Optimizations**. Turn on **Perform WYSIWYG primitive resynthesis (using optimization techniques specified in Analysis & Synthesis settings)**.

The Quartus II technology mapper optimizes the design for **Speed**, **Area**, or **Balanced**, according to the setting of the **Optimization Technique** option. To change this setting, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and choose **Speed** or **Balanced** under **Optimization Technique**.

The **Perform gate-level register retiming** option enables movement of registers across combinational logic to balance timing, allowing the Quartus II software to balance the delay between timing-critical paths and non-critical paths. You can use this option with Quartus II integrated synthesis, or if you are using a third-party EDA synthesis tool, you can use this option if you have turned on **Perform WYSIWYG primitive resynthesis (using optimization techniques specified in Analysis & Synthesis settings)**. After you turn on the **Perform gate-level register retiming**, you can optionally check the Trade off t_{CO}/t_{SU} with f_{MAX} setting. When you turn on this setting, register retiming can affect registers that are connected to the I/O pins. If this setting is not turned on, the Quartus II software does not touch any of the registers that are connected to the I/O pins when doing register retiming.

The physical synthesis optimizations occur during the Fitter stage of Quartus II compilation. Physical synthesis optimizations make placement-specific changes to the netlist that improve speed performance results for a specific Altera device.

The following physical synthesis optimizations are available during the trigger stage for improving performance:

- Physical synthesis for combinational logic
- Automatic asynchronous signal pipelining
- Physical synthesis for registers
  - Register duplication
  - Register retiming

On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**, and specify the physical synthesis optimization options on the **Physical Synthesis Optimizations** page. You can also specify the **Physical synthesis effort**, which sets the level of physical synthesis optimization that you want the Quartus II software to perform.

The **Perform physical synthesis for combinational logic** option allows the Quartus II Fitter to resynthesize the combinational logic in a design to reduce delay along the critical path and improve design performance.

The **Perform automatic asynchronous signal pipelining** option allows the Quartus II Fitter to insert pipeline stages for asynchronous clear and asynchronous load signals automatically during fitting to increase circuit

performance. You can use this option if asynchronous control signal recovery and removal times do not meet your requirements. The option improves performance for designs in which asynchronous signals in very fast clock domains cannot be distributed across the chip quickly enough (because of long global network delays).



The **Perform automatic asynchronous signal pipelining** option adds registers to nets driving the asynchronous clear or asynchronous load ports of registers. This adds register delays (and latency) to the reset, adding the same number of register delays for each destination using the reset. Therefore, the option should be used only when adding latency to reset signals does not violate any design requirements. This option also prevents the promotion of signals to use global routing resources.

The **Perform register duplication** Fitter option allows the Quartus II Fitter to duplicate registers based on Fitter placement information to improve design performance. The Fitter can also duplicate combinational logic when this option is enabled.

The **Perform register retiming** Fitter option allows the Quartus II Fitter to move registers across combinational logic to balance timing. This option turns on algorithms similar to the **Perform gate-level register retiming** option. This option applies to registers and combinational logic that have already been placed into logic cells, and it compliments the synthesis gate-level option.



For more information and detailed descriptions of these netlist optimization options, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

Because performance results are design-dependent, try these options in different combinations until you achieve the best results. Generally, turning on all the options gives the best results but significantly increases compilation time. This section provides typical benchmark results on different designs with varying amounts of logic using synthesis netlists from leading third-party synthesis tools and compiled with Quartus II software. These results use the default Balanced setting for the Optimization Technique for WYSIWYG resynthesis. Changing the setting to Speed or Area can affect your results.

Tables 8–6 through 8–8 show the average results for different device families, using the following measures for the quality of results:

- **f_{MAX} Gain**—The percentage gain in clock f_{MAX} performance
- **Win Ratio**—The percentage of designs that showed better performance with the option on than without the option on

- **Winner's f_{MAX} Gain**—The average percentage improvement for the designs that showed better performance with these settings (the designs considered a win)
- **Logic Area Change**—The percentage gain in logic utilization. Negative values mean reduced area; positive values mean increased area
- **Compile Time Change**—The multiplication factor for the compilation time when the option is used

Table 8–6. Average Results of Synthesis Netlist and Physical Synthesis Optimizations for Stratix and Cyclone Designs

Optimization Method	f_{MAX} Gain (%)	Win Ratio (%)	Winner's f_{MAX} Gain (%)	Logic Area Change (%)	Compile Time Change (×)
WYSIWYG primitive resynthesis	3	60	6	–8	1.0
Physical synthesis for combinational logic and registers					
Using physical synthesis Fast effort level	10	86	12	4	1.4
Using physical synthesis Normal effort level	15	86	16	4	2.2
Using physical synthesis Extra effort level	17	86	18	4	3.7

Table 8–7. Average Results of Synthesis Netlist and Physical Synthesis Optimizations for Arria GX, Stratix II, Stratix II GX, and Cyclone II Designs

Optimization Method	f_{MAX} Gain (%)	Win Ratio (%)	Winner's f_{MAX} Gain (%)	Logic Area Change (%)	Compile Time Change (×)
WYSIWYG primitive resynthesis	3	60	6	–8	1.0
Physical synthesis for combinational logic and registers					
Using physical synthesis Fast effort level	11	82	14	2.5	1.3
Using physical synthesis Normal effort level	14	88	17	4	2.0
Using physical synthesis Extra effort level	15	88	18	4.3	2.3

Table 8–8. Average Performance of Synthesis Netlist and Physical Synthesis Optimizations for Stratix III and Cyclone III Designs

Optimization Method	f_{MAX} Gain (%)	Win Ratio (%)	Winner's f_{MAX} Gain (%)	Logic Area Change (%)	Compile Time Change (×)
WYSIWYG primitive resynthesis	3	60	6	–8	1.0
Physical synthesis for combinational logic and registers					
Using physical synthesis Fast effort level	8	75	11	2	1.2
Using physical synthesis Normal effort level	10	79	14	3	1.7
Using physical synthesis Extra effort level	11	79	15	3	2.0

Turn Off Extra-Effort Power Optimization Settings

If PowerPlay Power Optimization settings are set to **Extra Effort**, your design performance may be affected. If improving timing performance is more important than reducing power use, set the Power Optimization setting to **Normal**.

To change the PowerPlay Power Optimization level, on the Assignments menu, choose **Settings**. The Setting dialog box appears. From the **Category** list, select **Analysis & Synthesis Settings**. From the drop-down menu, select the appropriate level of PowerPlay Power Optimization level.



For more information about reducing power use, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Optimize Synthesis for Speed, Not Area

The manner in which the design is synthesized has a large impact on design performance. Design performance varies depending on the way the design is coded, the synthesis tool used, and the options specified when synthesizing. Change your synthesis options if a large number of paths are failing, or if specific paths are failing badly and have many levels of logic.

Set your device and timing constraints in your synthesis tool. Synthesis tools are timing-driven and optimized to meet specified timing requirements. If you do not specify target frequency, some synthesis tools optimize for area.

Some synthesis tools offer an easy way to instruct the tool to focus on speed instead of area.

For Quartus II integrated synthesis, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and specify **Speed** as the **Optimization Technique** option. You can also specify this logic option for specific modules in your design with the Assignment Editor while leaving the default **Optimization Technique** setting at **Balanced** (for the best trade-off between area and speed for certain device families) or **Area** (if area is an important concern). You can also use the **Speed Optimization Technique for Clock Domains** option to specify that all combinational logic in or between the specified clock domain(s) is optimized for speed.

To achieve best performance with push-button compilation, follow the recommendations in the following sections for other synthesis settings. You can use DSE to experiment with different Quartus II synthesis options to optimize your design for the best performance.



For information about setting timing requirements and synthesis options in Quartus II integrated synthesis and third-party synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*, or refer to your synthesis software documentation.

Flatten the Hierarchy During Synthesis

Synthesis tools typically let you preserve hierarchical boundaries, which can be useful for verification or other purposes. However, the best optimization results generally occur when the synthesis tool optimizes across hierarchical boundaries because doing so often allows the synthesis tool to perform the most logic minimization, which can improve performance. Whenever possible, flatten your design hierarchy to achieve the best results. If you are using Quartus II integrated synthesis, ensure that the **Preserve Hierarchical Boundary** option is turned off. If you are using Quartus II incremental compilation, you cannot flatten your design across design partitions. Incremental compilation always preserves the hierarchical boundaries between design partitions. Follow Altera's recommendations for design partitioning such as registering partition boundaries to reduce the effect of cross-boundary optimizations.



For more information about using incremental compilation and recommendations for design partitioning, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. If you are using an incremental synthesis flow that requires separate hierarchy blocks, you can find additional recommendations for design partitioning in the *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*.

Set the Synthesis Effort to High

Some synthesis tools offer varying synthesis effort levels to trade off compilation time with synthesis results. Set the synthesis effort to high to achieve best results when applicable.

Change State Machine Encoding

State machines can be encoded using various techniques. One-hot encoding, which uses one register for every state bit, usually provides the best performance. If your design contains state machines, changing the state machine encoding to one-hot can improve performance at the cost of area.

If your design does not manually encode the state bits, you can select the state machine encoding chosen in your synthesis tool. In Quartus II integrated synthesis, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and for **State Machine Processing,** choose **One-Hot**. You also can specify this logic option for specific modules or state machines in your design with the Assignment Editor.

In some cases (especially in Stratix II and Stratix III devices), encoding styles other than the default offer better performance. Experiment with different encoding styles to see what effect the style has on your resource utilization and timing performance.

Duplicate Logic for Fan-Out Control

Duplicating logic or registers can help improve timing in cases where moving a register in a failing timing path to reduce routing delay creates other failing paths, or where there are timing problems due to the fan-out of the registers.

Many synthesis tools support options or attributes that specify the maximum fan-out of a register. When using Quartus II integrated synthesis, you can set the **Maximum Fan-Out** logic option in the Assignment Editor to control the number of destinations for a node so that the fan-out count does not exceed a specified value. You can also use the `maxfan` attribute in your HDL code. The software duplicates the node as needed to achieve the specified maximum fan-out.



Logic duplication using Maximum Fan-Out assignments normally increases resource utilization, and can potentially increase compilation time, depending on the placement and the total resource usage within the selected device. The improvement in timing performance that results because of Maximum Fan-Out assignments is very design-specific.

If you are using Maximum Fan-Out assignments, Altera recommends benchmarking your design with and without these assignments to evaluate whether they give the expected improvement in timing performance, and use the assignments only when you get improved results.

You can manually duplicate registers in the Quartus II software regardless of the synthesis tool used. To duplicate a register, apply the **Manual Logic Duplication** option to the register with the Assignment Editor.

The manual logic duplication option also accepts wildcards. This is an easy and powerful duplication technique that you can use without editing your source code. You can use this technique, for example, to make a duplicate of a large fan-out node for all of its destinations in a certain design hierarchy, such as `hierarchy_A`. To apply such an assignment in the Assignment Editor, make an entry such as the one shown in Table 8–9:

Table 8–9. Duplicating Logic in the Assignment Editor

From	To	Assignment Name	Value
`My_high_fanout_node`	`*hierarchy_A*`	Manual Logic Duplication	`high_fanout_to_A`



For more information about the manual logic duplication option, refer to the Quartus II Help.

Prevent Shift Register Inference

In some cases, turning off the inference of shift registers increases performance. Doing so forces the software to use logic cells to implement the shift register instead of implementing the registers in memory blocks using the `altshift_taps` megafunction. If you implement shift registers in logic cells instead of memory, logic utilization is increased.

Use Other Synthesis Options Available in Your Synthesis Tool

With your synthesis tool, experiment with the following options if they are available:

- Turn on register balancing or retiming
- Turn on register pipelining
- Turn off resource sharing

These options may increase performance. They typically increase the resource utilization of your design.

Fitter Seed

The Fitter seed affects the initial placement configuration of the design. Changing the seed value changes the Fitter results because the fitting results change whenever there is a change in the initial conditions. Because each seed value results in a somewhat different fit, you can experiment with several different seeds to attempt to obtain better fitting results and timing performance.

When there are changes in your design, there is some random variation in performance between compilations. This variation is inherent in placement and routing algorithms—there are too many possibilities to try them all and get the absolute best result, so the initial conditions change the compilation result.

Note that any design change that directly or indirectly affects the Fitter has the same type of random effect as changing the seed value. This includes any change in source files, Analysis and Synthesis settings, Fitter settings, or Timing Analyzer settings. The same effect can appear if you use a different computer processor type or different operating system because different systems can change the way floating point numbers are calculated in the Fitter.

If a change in optimization settings slightly affects the register-to-register timing or number of failing paths, you can't always be certain that your change caused the improvement or degradation or whether it could be due to random effects in the Fitter. If your design is still changing, running a seed sweep (compiling your design with multiple seeds) determines whether the average result has improved after an optimization change and whether a setting that increases compilation time has benefits worth the increased time (such as turning the Physical Synthesis Effort to **Extra**). The sweep also shows the amount of random variation you should expect for your design.

If your design is finalized, you can compile your design with different seeds to obtain one optimal result. However, if you subsequently make any changes to your design, you will likely have to perform seed sweep again.

On the Assignments menu, select **Fitter Settings** to control the initial placement with the Seed. You can use the Design Space Explorer (DSE) to perform a seed sweep easily.

You can use the following Tcl command from a script to specify a Fitter seed:

```
set_global_assignment -name SEED <value>
```



For more information about compiling with different seeds using the DSE script, refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*.

Optimize Source Code

If the methods described in the preceding sections do not sufficiently improve timing of the design, modify your design files to achieve the desired results. Try restructuring the design to use pipelining or more efficient coding techniques. In many cases, optimizing the design's source code can have a very significant effect on your design performance. In fact, optimizing your source code is typically the most effective technique for improving the quality of your results, and is often a better choice than using LogicLock or location assignments.

If the critical path in your design involves memory or DSP functions, check whether you have code blocks in your design that describe memory or functions that are not being inferred and placed in dedicated logic. You may be able to modify your source code to cause these functions to be placed into high-performance dedicated memory or resources in the target device.

Ensure that your state machines are recognized as state machine logic and optimized appropriately in your synthesis tool. State machines that are recognized are generally optimized better than if the synthesis tool treats them as generic logic. In the Quartus II software, you can check for the State Machine report under **Analysis & Synthesis** in the Compilation Report. This report provides details, including the state encoding for each state machine that was recognized during compilation. If your state machine is not being recognized, you may need to change your source code to enable it to be recognized.



For coding style guidelines including examples of HDL code for inferring memory, and functions and guidelines and sample HDL code for state machines, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

LogicLock Assignments

You can make LogicLock assignments for optimization based on nodes, design hierarchy, or critical paths. This method can be used if a large number of paths are failing, and recoding the design does not seem to be necessary. LogicLock assignments can help if routing delays form a large portion of your critical path delay, and placing logic closer together in the device improves the routing delay. This technique is most beneficial for devices with hierarchical routing structures such as the APEX 20K device family.



Improving fitting results with LogicLock assignments, especially for larger devices, such as the Stratix series of devices and Arria GX devices, can be difficult. The LogicLock feature is intended to be used for performance preservation; therefore, LogicLock assignments do not always improve the performance of the design. In many cases, you cannot improve upon results from the Fitter by making location assignments.

If there are existing LogicLock assignments in your design, remove the assignments if your design methodology permits it. Recompile the design to see if the assignments are making the performance worse.

When making LogicLock assignments, it is important to consider how much flexibility to give the Fitter. LogicLock assignments provide more flexibility than hard location assignments. Assignments that are more flexible require higher Fitter effort, but reduce the chance of design over-constraint. The following types of LogicLock assignments are available, listed in order of decreasing flexibility:

- Soft LogicLock regions
- Auto size, floating location regions
- Fixed size, floating location regions
- Fixed size, locked location regions

To determine what to put into a LogicLock region, refer to the timing analysis results and the Chip Planner (Timing Closure Floorplan for older devices). The register-to-register timing paths in the Timing Analyzer section of the Compilation Report helps you recognize patterns.

The following sections describe cases in which LogicLock regions can help to optimize a design.



For more information about using LogicLock regions, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Hierarchy Assignments

For a design with the hierarchy shown in Figure 8–11, which has failing paths in the timing analysis results similar to those shown in Table 8–10, `mod_A` is probably a problem module. In this case, a good strategy to fix the failing paths is to place the `mod_A` hierarchy block in a LogicLock region so that all the nodes are closer together in the floorplan.

Figure 8–11. Design Hierarchy

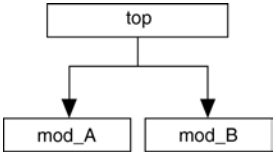


Table 8–10 shows the failing paths connecting two regions together within `mod_A` listed in the timing analysis report.

Table 8–10. Failing Paths in a Module Listed in Timing Analysis

From	To
|mod_A|reg1	|mod_A|reg9
|mod_A|reg3	|mod_A|reg5
|mod_A|reg4	|mod_A|reg6
|mod_A|reg7	|mod_A|reg10
|mod_A|reg0	|mod_A|reg2

Hierarchical LogicLock regions are also important if you are using an incremental compilation flow. Each design partition for incremental compilation should be placed in a separate LogicLock region to reduce conflicts and ensure good quality of results as the design develops. In this case, you should not use soft LogicLock regions because they allow the Fitter to move nodes away from the region. You can use auto size and floating location regions to find a good design floorplan, but you should then fix the size and placement to achieve the best results in future compilations.

For more information about using incremental compilation and recommendations for creating a design floorplan using LogicLock regions, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*, and *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Path Assignments

If you see a pattern such as the one shown in Figure 8–12 and Table 8–11, it often indicates paths with a common problem. In this case, a path-based assignment can be made from all `d_reg` registers to all `memaddr`

registers. You can make a path-based assignment to place all source registers, destination registers, and the nodes between them in a LogicLock region with the wildcard characters "*" and "?".

You also can explicitly place the nodes of a critical path in a LogicLock region. However, using this method instead of path assignments can result in alternate paths between the source and destination registers becoming critical paths.

Figure 8–12. Failing Paths in Timing Analysis

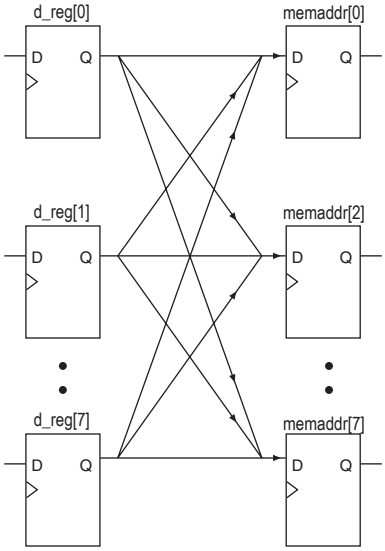


Table 8–11 shows the failing paths listed in the timing analysis report.

Table 8–11. Failing Paths in Timing Analysis

From	To
|d_reg[1]	|memaddr[5]
|d_reg[1]	|memaddr[6]
|d_reg[1]	|memaddr[7]
|d_reg[2]	|memaddr[0]
|d_reg[2]	|memaddr[1]



For more information about path-based LogicLock assignments, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Location Assignments and Back-Annotation

If a small number of paths are failing to meet their timing requirements, you can use hard location assignments to optimize placement. Location assignments are less flexible for the Quartus II Fitter than LogicLock assignments. In some cases, when you are very familiar with your design, you can enter location constraints in a way that produces better results.



Improving fitting results, especially for larger devices, such as the Stratix series of devices and Arria GX devices, can be difficult. Location assignments do not always improve the performance of the design. In many cases, you cannot improve upon the results from the Fitter by making location assignments.

The following are commonly used location assignments, listed in order of decreasing flexibility:

- Custom regions
- Back-annotated LAB location assignments
- Back-annotated LE or ALM location assignments

Custom Regions

A custom region is a rectangular region containing user-assigned nodes, which are constrained in the region's boundaries. If any portion of a block in the device floorplan, such as an M-RAM block, overlaps a custom region, it is considered to be entirely in that region.

Custom regions are hard location assignments that cannot be overridden and are very similar to fixed-size, locked-location, LogicLock regions. Custom regions are commonly used when logic must be constrained to a specific portion of the device.

Back-Annotation and Manual Placement

Assigning the location of nodes in a design to the locations to which they were assigned during the last compilation is called "back-annotation". When nodes are locked to their assigned locations in a back-annotated design, you can manually move specific nodes without affecting other back-annotated nodes. The process of manually moving and reassigning specific nodes is called manual placement.



Back-annotation is very restrictive to the compiler, so you should back-annotate only when the design has been finalized and no further changes are expected. Assignments can become invalid if the design is changed. Combinational nodes often change names when a design is resynthesized, even if they are unrelated to the logic that was changed.

Moving nodes manually can be very difficult for large devices. In many cases, you cannot improve upon the Fitter's results.

Illegal or unroutable location constraints can cause "no fit" errors.

Before making location assignments, determine whether to back-annotate to lock down the assigned locations of all nodes in the design. When you are using a hierarchical design flow, you can lock down node locations in one LogicLock region only, while other node locations are left floating in a fixed LogicLock region. By implementing a hierarchical approach, you can use the LogicLock design methodology to reduce the dependence of logic blocks on other logic blocks in the device.

Consistent node names are required to perform back-annotation. If you use Quartus II integrated synthesis or any Quartus II optimizations, such as the WYSIWYG primitive resynthesis netlist optimization or any physical synthesis optimizations, you must create an atom netlist before you back-annotate to lock down the placement of any nodes. This creates consistent node names.



Physical synthesis optimizations are placement-specific as well as design-specific. Unless you back-annotate the design before recompilation, the physical synthesis results can differ. This happens because the atom netlist creates different placement results. By back-annotating the design, the design source and the atom netlist use the same placement when the design is recompiled. When you are using an atom netlist and you want to maintain the same placement results as a previous compilation, use LogicLock regions and back-annotate the placement of all nodes in the design. Not back-annotating the design can result in the design source and the atom netlist having different placement results and therefore different synthesis results.



For more information about creating atom netlists for your design, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

When you back-annotate a design, you can choose whether to assign the nodes either to LABs (this is preferred because of increased flexibility) or LEs/ALMs. You also can choose to back-annotate routing to further restrict the Fitter and force a specific routing within the device.



Using back-annotated routing with physical synthesis optimizations can result in a routing failure.



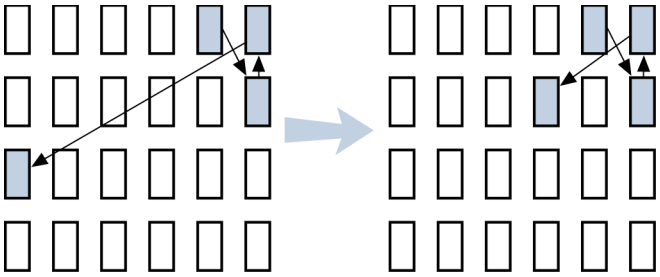
For more information about back-annotating routing, refer to the Quartus II Help.

When performing manual placement at a detailed level, Altera recommends that you move LABs, not logic cells (LEs or ALMs). The Quartus II software places nodes that share the same control signals in appropriate LABs. Successful placement and routing is more difficult when you move individual logic cells. This is because LEs with different control signals are put into the same LAB may not have any unused control signals available and the design may not fit.

In general, when you are performing manual placement and routing, fix all I/O paths first, because often fewer options are available to meet I/O timing. After I/O timing is met, focus on manually placing register-to-register timing paths. This strategy is consistent with the methodology outlined in this chapter.

The best way to meet performance is to move nodes closer together. For a critical path such as the one shown in Figure 8–13, moving the destination node closer to the other nodes reduces the delay and helps meet your timing requirements.

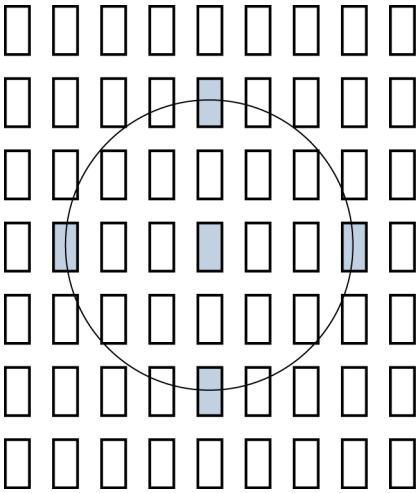
Figure 8–13. Reducing Delay of Critical Path



Optimizing Placement for Stratix, Stratix II, Arria GX, and Cyclone II Devices

In the Arria GX devices and Stratix and Cyclone series of devices, the row interconnect delay is slightly faster than the column interconnect delay. Therefore, when placing nodes, optimal placement is typically an ellipse around the source or destination node. In Figure 8–14, if the source is located in the center, any of the shaded LABs should give approximately the same delay.

Figure 8–14. Possible Optimal Placement Ellipse



In addition, you should avoid crossing any M-RAM memory blocks for node-to-node routing, because routing paths across M-RAM blocks requires using R24 or C16 routing lines.

The Quartus II software calculates the interconnect delay based on different electrical characteristics of each individual wire, such as the length, fan-out, distribution of the parasitic loading on the wire, and so forth.

To determine the actual delays to and from a resource, use the **Show Physical Timing Estimate** feature in the Chip Planner.

For more information about using the Chip Planner or the Timing Closure Floorplan, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Optimizing Placement for Cyclone Devices

In Cyclone devices, the row and column interconnect delays are similar; therefore, when placing nodes, optimal placement is typically a circle around the source or destination node.

Try to avoid long routes across the device. Long routes require more than one routing line to cross the Cyclone device.

Optimizing Placement for Mercury, APEX II, and APEX 20KE/C Devices

For the Mercury, APEX II, and APEX 20KE/C device families, the delay for paths is reduced by placing the source and destination nodes in the same geographical resource location. The following list shows the device resources, listed in order from fastest to slowest:

- LAB
- MegaLAB structure
- MegaLAB column
- Row

For example, if the nodes cannot be placed in the same MegaLAB structure to reduce the delay, place them in the same MegaLAB column. For the actual delays to and from resources, use the **Show Physical Timing Estimate** feature in the Timing Closure Floorplan.

Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)

The following recommendations help you take advantage of the macrocell-based architecture in the MAX 7000 and MAX 3000 device families to yield maximum speed, reliability, and device resource utilization while minimizing fitting difficulties.

After design analysis, the first stage of design optimization is to improve resource utilization. Complete this stage before proceeding to timing optimization. First, ensure that you have set the basic constraints described in "Initial Compilation: Required Settings" on page 8–3. If your design is not fitting into a specified device, use the techniques in this section to achieve a successful fit.

Use Dedicated Inputs for Global Control Signals

MAX 7000 and MAX 3000 devices have four dedicated inputs that can be used for global register control. Because the global register control signals can bypass the logic cell array and directly feed registers, product terms can be preserved for primary logic. Also, because each signal has a dedicated path into the LAB, global signals also can bypass logic and data path interconnect resources.

Because the dedicated input pins are designed for high fan-out control signals and provide low skew, you should always assign global signals (such as clock, clear, and output enable) to the dedicated input pins.

You can use logic-generated control signals for global control signals instead of dedicated inputs. However, the following list shows the disadvantages to using logic-generated control signals:

- More resources are required (logic cells, interconnect).
- More data skew is introduced.
- If the logic-generated control signals have high fan-out, the design may be more difficult to fit.

By default, the Quartus II software uses dedicated inputs for global control signals automatically. You can assign control signals to dedicated input pins in one of the following ways:

- In the Assignment Editor, choose one of the two following methods:
  - Assign pins to dedicated pin locations.
  - Assign a **Global Signal** setting to the pins.
- On the Assignments menu, click **Settings**. On the **Analysis & Synthesis Settings** page, in the **Auto Global Options** section, in the **Category** list, select **Register Control Signals**.
- Insert a `GLOBAL` primitive after the pins.
- If you have already assigned pins for the design in the MAX+PLUS® II software, on the Assignments menu, click **Import Assignments**.

Reserve Device Resources

Because pin and logic option assignments may be necessary for board layout and performance requirements, and because full utilization of the device resources can increase the difficulty of fitting the design, Altera recommends that you leave 10% of the device's logic cells and 5% of the I/O pins unused to accommodate future design modifications. Following the Altera-recommended device resource reservation guidelines for macrocell-based CPLDs increases the chance that the Quartus II software can fit the design during recompilation after changes or assignments have been made.

Pin Assignment Guidelines and Procedures

Sometimes user-specified pin assignments are necessary for board layout. This section discusses pin assignment guidelines and procedures.

To minimize fitting issues with pin assignments, follow these guidelines:

- Assign speed-critical control signals to dedicated inputs.
- Assign output enables to appropriate locations.
- Estimate fan-in to assign output pins to the appropriate LAB.
- Assign output pins that require parallel expanders to macrocells numbered 4 to 16.



Altera recommends that you allow the Quartus II software to choose pin assignments automatically when possible.

Control Signal Pin Assignments

Assign speed-critical control signals to dedicated input pins. Every MAX 7000 and MAX 3000 device has four dedicated input pins (GCLK1, OE2/GCLK2, OE1, and GCLRn). You can assign clocks to global clock dedicated inputs (GCLK1, and OE2/GCLK2), clear to the global clear dedicated input (GCLRn), and speed-critical output enable to global OE dedicated inputs (OE1, and OE2/GCLK2).

Output Enable Pin Assignments

Occasionally, because the total number of required output enable pins is more than the dedicated input pins, output enable signals must be assigned to I/O pins.



To minimize possible fitting errors when assigning the output enable pins for MAX 7000 and MAX 3000 devices, refer to Pin-Out Files for Altera Devices on the Altera website **(www.altera.com)**.

Estimate Fan-In When Assigning Output Pins

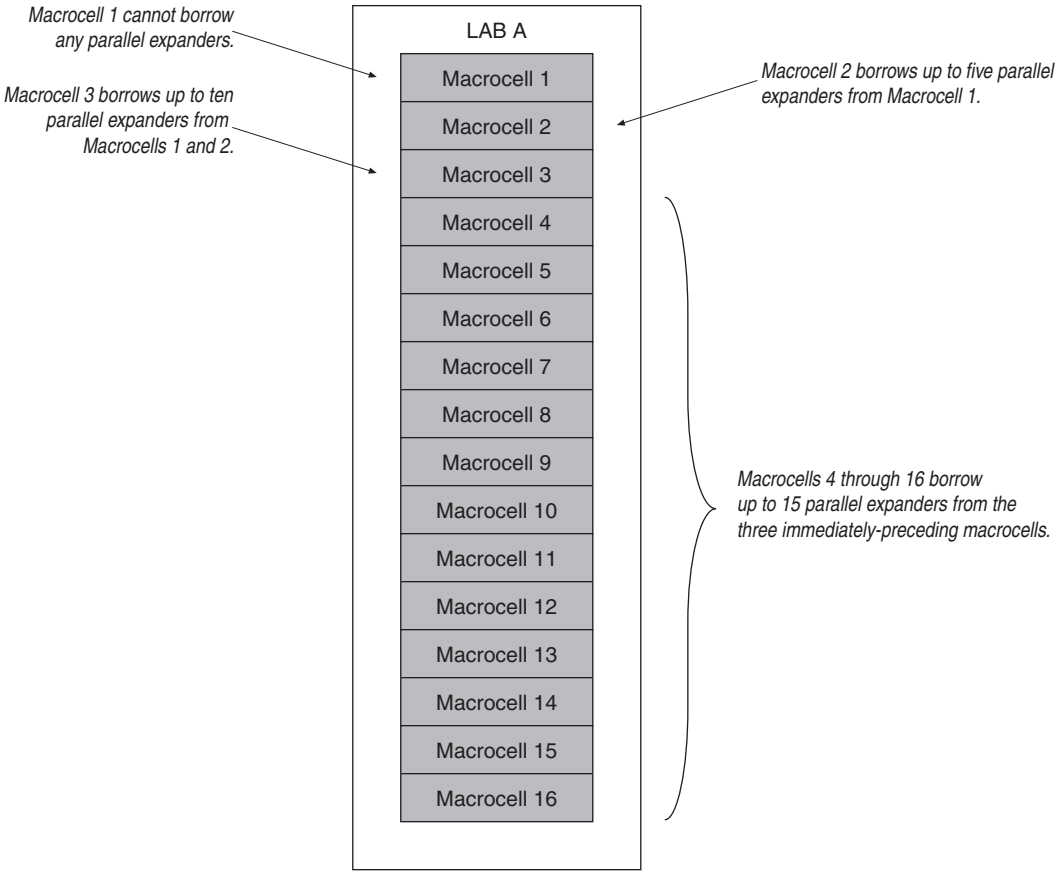
Macrocells with high fan-in can cause more placement problems for the Quartus II Fitter than those with low fan-in. The maximum fan-in per LAB should not exceed 36 in MAX 7000 and MAX 3000 devices. Therefore, estimate the fan-in of logic (such as an *x*-input AND gate) that feeds each output pin. If the total fan-in of logic that feeds each output pin in the same LAB exceeds 36, compilation may fail. To save resources and prevent compilation errors, avoid assigning pins that have high fan-in.

Outputs Using Parallel Expander Pin Assignments

Figure 8–15 illustrates how parallel expanders are used within a LAB. MAX 7000 and MAX 3000 devices contain chains that can lend or borrow parallel expanders. The Quartus II Fitter places macrocells in a location that allows them to lend and borrow parallel expanders appropriately.

As shown in Figure 8–15, only macrocells 2 through 16 can borrow parallel expanders. Therefore, assign output pins that may need parallel expanders to pins adjacent to macrocells 4 through 16. Altera recommends using macrocells 4 through 16 because they can borrow the largest number of parallel expanders.

Figure 8–15. LAB Macrocells and Parallel Expander Associations



Resolving Resource Utilization Problems

Two common Quartus II compilation fitting issues cause errors: excessive macrocell usage and lack of routing resources. Macrocell usage errors occur when the total number of macrocells in the design exceeds the available macrocells in the device. Routing errors occur when the available routing resources are insufficient to implement the design. Check the Message window for the compilation results.



Messages in the Messages window are also copied in the Report Files. Right-click on a message and select **Help** for more information.

Resolving Macrocell Usage Issues

Occasionally, a design requires more macrocell resources than are available in the selected device, which results in the design not fitting. The following list provides tips for resolving macrocell usage issues as well as tips to minimize the number of macrocells used.

- On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and turn off **Auto Parallel Expanders**. If the design's clock frequency (f_{MAX}) is not an important design requirement, turn off parallel expanders for all or part of the project. The design usually requires more macrocells if parallel expanders are turned on.
- Change Optimization Technique from **Speed** to **Area**. Selecting **Area** instructs the compiler to give preference to area utilization rather than speed (f_{MAX}). On the Assignments menu, click **Settings**. In the **Category** list, change the **Optimization Technique** option in the **Analysis & Synthesis Settings** page.
- Use D-type flipflops instead of latches. Altera recommends that you always use D-type flipflops instead of latches in your design because D-type flipflops can reduce the macrocell fan-in, and thus reduce macrocell usage. The Quartus II software uses extra logic to implement latches in MAX 7000 and MAX 3000 designs because MAX 7000 and MAX 3000 macrocells contain D-type flipflops instead of latches.
- Use asynchronous clear and preset instead of synchronous clear and preset. To reduce the product term usage, use asynchronous clear and preset in your design whenever possible. Using other control signals such as synchronous clear produces macrocells and pins with higher fan-out.



After following the suggestions in this section, if your project still does not fit the targeted device, consider using a larger device. When upgrading to a different density, the vertical package-migration feature of the MAX 7000 and MAX 3000 device families allows pin assignments to be maintained.

Resolving Routing Issues

Routing is another resource that can cause design fitting issues. For example, if the total fan-in into a LAB exceeds the maximum allowed, a no-fit error can occur during compilation. If your design does not fit the targeted device because of routing issues, consider the following suggestions.

- Use dedicated inputs/global signals for high fan-out signals. The dedicated inputs in MAX 7000 and MAX 3000 devices are designed for speed-critical and high fan-out signals. Always assign high fan-out signals to dedicated inputs/global signals.
- Change the **Optimization Technique** option from **Speed** to **Area**. This option may resolve routing resource and macrocell usage issues. Refer to the same suggestion in "Resolving Macrocell Usage Issues" on page 8–75.
- Reduce the fan-in per cell. If you are not limited by the number of macrocells used in the design, you can use the **Fan-in per cell (%)** option to reduce the fan-in per cell. The allowable values are 20–100%; the default value is 100%. Reducing the fan-in can reduce localized routing congestion but increase the macrocell count. You can set this logic option in the Assignment Editor or under **More Settings** in the **Analysis & Synthesis Settings** page of the **Settings** dialog box.
- On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and turn off **Auto Parallel Expanders**. By turning off the parallel expanders, you give Quartus II software more fitting flexibility for each macrocell, allowing macrocells to be relocated. For example, each macrocell (previously grouped together in the same LAB) can be moved to a different LAB to reduce routing constraints.

- Insert logic cells. Inserting logic cells reduces fan-in and shared expanders used per macrocell, increasing routability. By default, the Quartus II software automatically inserts logic cells when necessary. Otherwise, **Auto Logic Cell** can be disabled as follows. On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**. Under More Settings, turn off **Auto Logic Cell Insertion**. Refer to "Using LCELL Buffers to Reduce Required Resources" on page 8–77 for more information.
- Change pin assignments. If you are willing to discard your pin assignments, you can let the Quartus II Fitter ignore some or all the assignments.



If you prefer reassigning pins to increase routing efficiency, refer to "Pin Assignment Guidelines and Procedures" on page 8–72.

Using LCELL Buffers to Reduce Required Resources

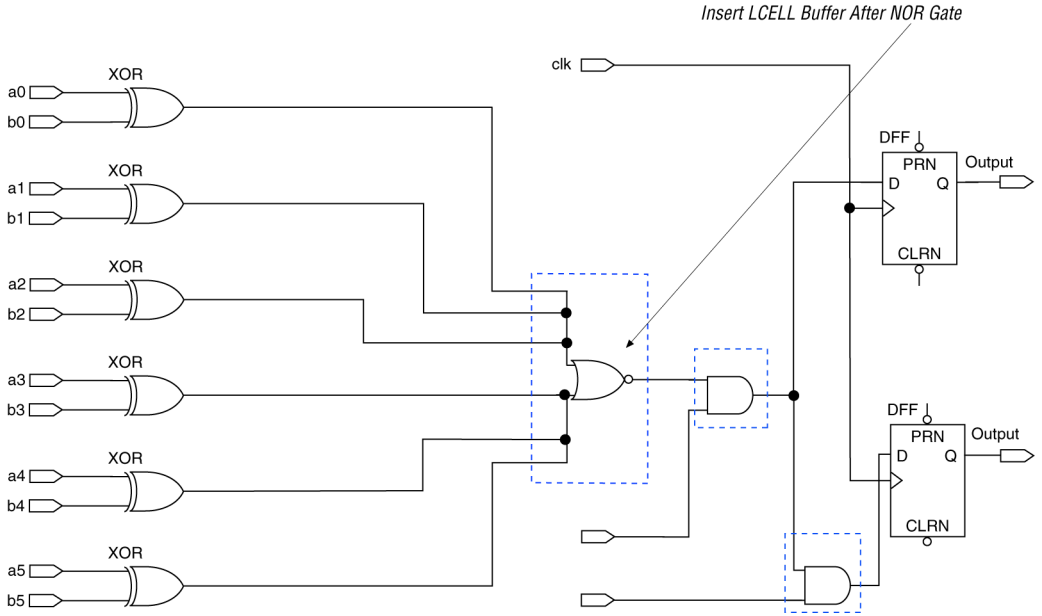
Complex logic, such as multilevel XOR gates, are often implemented with more than one macrocell. When this occurs, the Quartus II software automatically allocates shareable expanders—or additional macrocells (called synthesized logic cells)—to supplement the logic resources that are available in a single macrocell. You also can break down complex logic by inserting logic cells in the project to reduce the average fan-in and the total number of shareable expanders needed. Manually inserting logic cells can provide greater control over speed-critical paths.

Instead of using the Quartus II software's **Auto Logic Cell Insertion** option, you can manually insert logic cells. However, Altera recommends that you use the **Auto Logic Cell Insertion** option unless you know which part of the design is causing the congestion.

A good location to manually insert LCELL buffers is where a single complex logic expression feeds multiple destinations in your design. You can insert an LCELL buffer just after the complex expression; the Quartus II Fitter extracts this complex expression and places it in a separate logic cell. Rather than duplicate all the logic for each destination, the Quartus II software feeds the single output from the logic cell to all destinations.

To reduce fan-in and prevent no-fit compilations caused by routing resource issues, insert an LCELL buffer after a NOR gate (Figure 8–16). The design in Figure 8–16 was compiled for a MAX 7000AE device. Without the LCELL buffer, the design requires two macrocells and eight shareable expanders, and the average fan-in is 14.5 macrocells. However, with the LCELL buffer, the design requires three macrocells and eight shareable expanders, and the average fan-in is just 6.33 macrocells.

Figure 8–16. Reducing the Average Fan-In by Inserting LCELL Buffers



Timing Optimization Techniques (Macrocell-Based CPLDs)

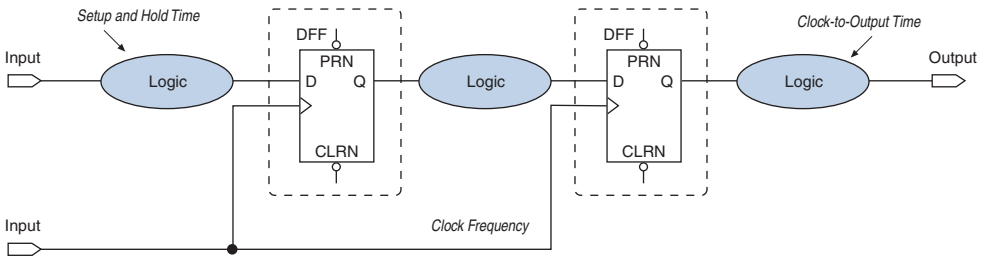
After resource optimization, design optimization focuses on timing. Ensure that you have made the appropriate assignments as described in "Initial Compilation: Required Settings" on page 8–3, and that the resource utilization is satisfactory before proceeding with timing optimization.

Maintaining system performance at or above certain timing requirements is an important goal of circuit designs. The following five timing parameters are primarily responsible for a design's performance:

- Setup time (t_{SU}), the propagation time for input data signals
- Hold time (t_H), the propagation time for input data signals
- Clock-to-output time (t_{CO}), the propagation time for output signals
- Pin-to-pin delays (t_{PD}), the time required for a signal from an input pin to propagate through combinational logic and appear at an external output pin
- Maximum clock frequency (f_{MAX}), the internal register-to-register performance

This section provides guidelines to improve the timing if the timing requirements are not met. Figure 8–17 shows the parts of the design that determine the t_{SU}, t_H, t_{CO}, t_{PD}, and f_{MAX} timing parameters.

Figure 8–17. Main Timing Parameters that Determine the System's Performance



Timing results for t_{SU}, t_H, t_{CO}, t_{PD}, and f_{MAX} are found in the Compilation Report for the Quartus II Classic Timing Analyzer, as discussed in "Design Analysis" on page 8–14.

When you are analyzing a design to improve performance, be sure to consider the two major contributors to long delay paths:

- Excessive levels of logic
- Excessive loading (high fan-out)

When a MAX 7000 or MAX 3000 device signal drives more than one LAB, the programmable interconnect array (PIA) delay increases by 0.1 ns per additional LAB fan-out. Therefore, to minimize the added delay, concentrate the destination macrocells into fewer LABs, minimizing the number of LABs that are driven. The main cause of long delays in circuit design is excessive levels of logic.

Improving Setup Time

Sometimes the t_{SU} timing reported by the Quartus II Fitter does not meet your timing requirements. To improve the t_{SU} timing, refer to the following guidelines:

- Turn on the **Fast Input Register** option using the Assignment Editor. The **Fast Input Register** option allows input pins to directly drive macrocell registers via the fast-input path, thus minimizing the pin-to-register delay. This option is useful when a pin drives a D-type flipflop and there is no combinational logic between the pin and the register.
- Reduce the amount of logic between the input and the register. Excessive logic between the input pin and register causes more delays. To improve setup time, Altera recommends that you reduce the amount of logic between the input pin and the register whenever possible.
- Reduce fan-out. The delay from input pins to macrocell registers increases when the fan-out of the pins increases. To improve the setup time, minimize the fan-out.

Improving Clock-to-Output Time

To improve a design's clock-to-output time, minimize the register-to-output-pin delay. To improve the t_{CO} timing, refer to the following guidelines.

- Use the global clock. In addition to minimizing the delay from a register to an output pin, minimizing the delay from the clock pin to the register can also improve t_{CO} timing. Always use the global clock for low-skew and speed-critical signals.
- Reduce the amount of logic between the register and output pin. Excessive logic between the register and the output pin causes more delay. Always minimize the amount of logic between the register and output pin for faster clock-to-output time.

Table 8–12 shows the timing results for an EPM7064AETC100-4 device when a combination of the **Fast Input Register** option, global clock, and minimal logic is used. When the **Fast Input Register** option is turned on, the t_{SU} timing is improved (t_{SU} decreases from 1.6 ns to 1.3 ns and from 2.8 ns to 2.5 ns). The t_{CO} timing is improved when the global clock is used for low-skew and speed-critical signals (t_{CO} decreases from 4.3 ns to 3.1 ns). However, if there is additional logic used between the input pin and the register or the register and the output pin, the t_{SU} and t_{CO} delays increase.

Table 8–12. EPM7064AETC100-4 Device Timing Results

Number of Registers	t_{SU} (ns)	t_H (ns)	t_{CO} (ns)	Global Clock Used	Fast Input Register Option	D Input Location	Q Output Location	Additional Logic Between:	
								D Input Location & Register	Register & Q Output Location
1	1.3	1.2	4.3	—	On	LAB A	LAB A	—	—
1	1.6	0.3	4.3	—	Off	LAB A	LAB A	—	—
1	2.5	0	3.1	✓	On	LAB A	LAB A	—	—
1	2.8	0	3.1	✓	Off	LAB A	LAB A	—	—
1	3.6	0	3.1	✓	Off	LAB A	LAB A	✓	—
1	2.8	0	7.0	✓	Off	LAB D	LAB A	—	✓
16 with the same D and clock inputs	2.8	0	All 6.2	✓	Off	LAB D	LAB A, B	—	—
32 with the same D and clock inputs	2.8	0	All 6.4	✓	Off	LAB C	LAB A, B, C	—	—

Improving Propagation Delay (t_{PD})

Achieving fast propagation delay (t_{PD}) timing is required in many system designs. However, if there are long delay paths through complex logic, achieving fast propagation delays can be difficult. To improve your design's t_{PD}, refer to the following guidelines.

- On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and turn on **Auto Parallel Expanders**. Turning on the parallel expanders for individual nodes or sub-designs can increase the performance of complex logic functions. However, if the project's pin or logic cell assignments use parallel expanders placed physically together with macrocells (which can reduce routability), parallel expanders can cause the Quartus II Fitter to have difficulties finding and optimizing a fit. Additionally, the number of macrocells required to implement the design increases and results in a no-fit error during compilation if the device resources are limited. For more information about turning the **Auto Parallel Expanders** option on, refer to "Resolving Macrocell Usage Issues" on page 8–75.
- Set the Optimization Technique to **Speed**. By default, the Quartus II software sets the **Optimization Technique** option to **Speed** for MAX 7000 and MAX 3000 devices. Reset the **Optimization Technique** option to **Speed** only if you previously set it to **Area**. On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and turn on **Speed** under **Optimization Technique**.

Improving Maximum Frequency (f_{MAX})

Maintaining the system clock at or above a certain frequency is a major goal in circuit design. For example, if you have a fully synchronous system that must run at 100 MHz, the longest delay path from the output of any register to the inputs of the registers it feeds must be less than 10 ns. Maintaining the system clock speed can be difficult if there are long delay paths through complex logic. Altera recommends that you follow the following guidelines to improve your design's clock speed (f_{MAX}).

- On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings** and turn on **Auto Parallel Expanders**. Turning on the parallel expanders for individual nodes or subdesigns can increase the performance of complex logic functions. However, if the project's pin or logic cell assignments use parallel expanders placed physically together with macrocells (which can reduce routability), parallel expanders can cause the Quartus II compiler to have difficulties finding and optimizing a fit. Additionally, the number of macrocells required to implement the design also increases and can result in a no-fit error during compilation if the device's resources are limited. For more information about using the **Auto Parallel Expanders** option, refer to "Resolving Macrocell Usage Issues" on page 8–75.
- Use global signals or dedicated inputs. Altera MAX 7000 and MAX 3000 devices have dedicated inputs that provide low skew and high speed for high fan-out signals. Minimize the number of control signals in the design and use the dedicated inputs to implement them.
- Set the **Optimization Technique** to **Speed**. By default, the Quartus II software sets the **Optimization Technique** option to **Speed** for MAX 7000 and MAX 3000 devices. Reset the **Optimization Technique** option to **Speed** only if you have previously set it to **Area**. You can reset the **Optimization Technique** option. In the **Category** list, choose **Analysis & Synthesis Settings**, and turn on **Speed** under Optimization Technique.
- Pipeline the design. Pipelining, which increases clock frequency (f_{MAX}), refers to dividing large blocks of combinational logic by inserting registers. For more information about pipelining, refer to "Optimizing Source Code—Pipelining for Complex Register Logic" on page 8–83.

Optimizing Source Code—Pipelining for Complex Register Logic

If the methods described in the preceding sections do not sufficiently improve your results, modify the design at the source to achieve the desired results. Using a pipelining technique can consume device resources, but it also lowers the propagation delay between registers, allowing you to maintain high system clock speed.

The benefits of pipelining can be demonstrated with a 4-to-16 pipelined decoder that decodes 4-bit numbers. The decoder is based on five 2-to-4 pipelined decoders with outputs that are registered using D-type flipflops. Figure 8–18 shows one of the 2-to-4 pipelined decoders. The function `2TO4DEC` is the 2-to-4 decoder that feeds all four decoded outputs (`out1`, `out2`, `out3`, and `out4`) to the D-type flipflops in `4REG`.

Figure 8–18. A 2- to 4-Pipelined Decoder

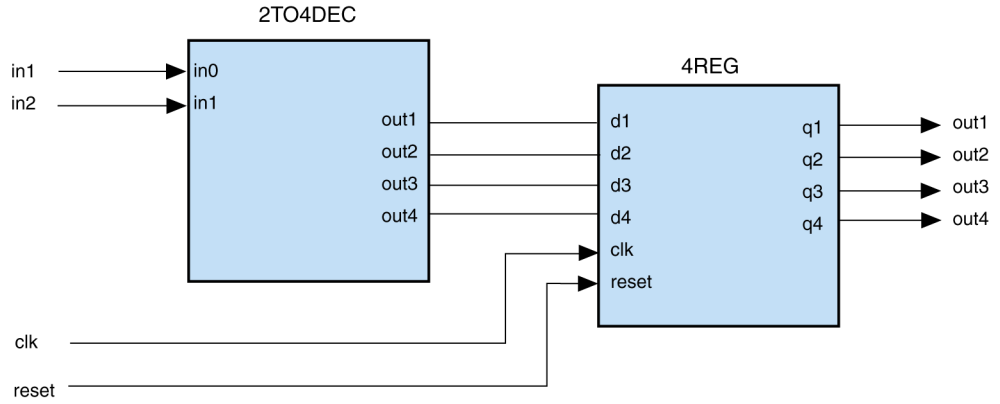
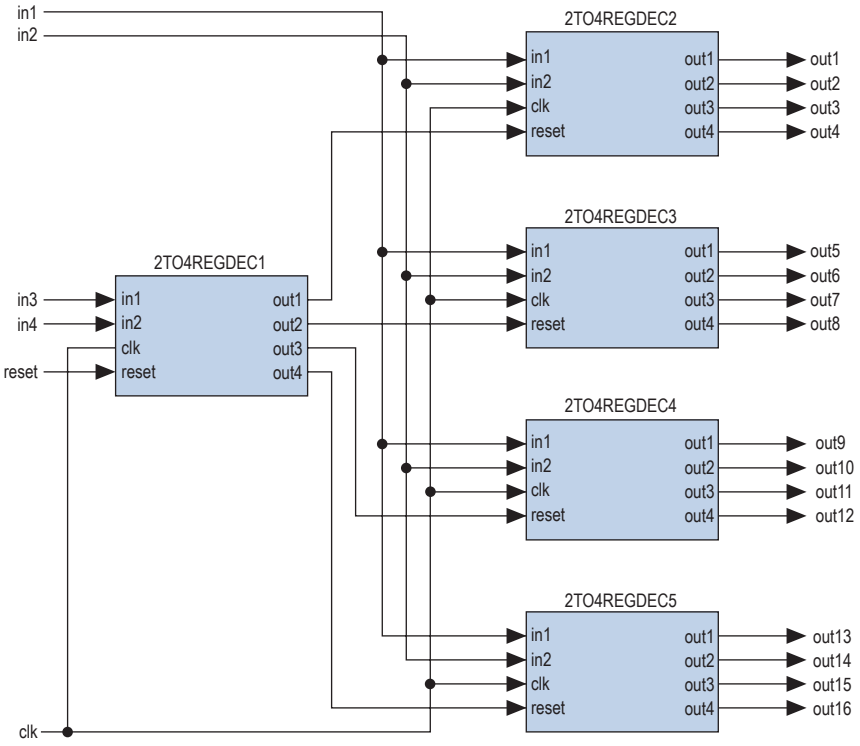


Figure 8–19 shows five 2-to-4 decoders (`2TO4REGDEC`) that are combined to form a 4-to-16 pipelined decoder. The first decoder (`2TO4REGDEC1`) decodes the two most significant bits (MSB) (`in3` and `in4`) of the 4-to-16 decoder. The decoded output from the `2TO4REGDEC1` decoder enables only one of the rest of the 2-to-4 decoders (`2TO4REGDEC2`, `2TO4REGDEC3`, `2TO4REGDEC4`, or `2TO4REGDEC5`). The inputs `in1` and `in2` are decoded by the enabled 2-to-4 decoder. Because the time to generate the decoded output increases with the size of the decoder, pipelining the design reduces the time consumed to generate the decoded output, thus improving the maximum frequency. In Figure 8–19, the MSBs (`in3` and `in4`) are decoded in the first clock cycle, while the other bits (`in1` and `in2`) are decoded in the following clock cycle.

Figure 8–19. Five 2-to-4 Pipelined Decoders Combined to Form a 4-to-16 Pipelined Decoder Note (1)



Note to Figure 8–19:
(1) This figure assumes an active low reset.

Compilation-Time Optimization Techniques

If reducing the compilation time of your design is important, use the techniques in this section. Be aware that reducing compilation time using some of these techniques can reduce the overall quality of results. A Compilation Time Advisor is also available in the Quartus II software, which helps you to reduce the compilation time. You can run the Compilation Time Advisor by pointing to **Advisors** on the **Tools** menu, and then clicking on the **Compilation Time Advisor**. You can find all the compilation time optimizing techniques described in this section in the Compilation Time Advisor as well.

If you open the Compilation Time Advisor after compilation, it displays recommendations on settings that can reduce the compilation time. Some of the recommendations from different advisors may contradict each other; Altera recommends evaluating the options, and choosing the settings that best suit your design requirements.

Incremental Compilation

You can speed up design iteration time by an average of 60% when making changes to the design and reach design timing closure more efficiently with the incremental compilation feature. Using incremental compilation allows you to organize your design into logical and physical partitions for design synthesis and fitting. Design iterations can be made dramatically faster by recompiling only a particular design partition and merging results with previous compilation results from other partitions. You can also use physical synthesis optimization techniques for specific design partitions while leaving other modules untouched to preserve performance.

When making changes to the design, use the incremental synthesis feature (part of incremental compilation) to save synthesis time. Incremental synthesis allows you to set design partitions to ensure that only those sections of a design that have been updated are resynthesized when the design is compiled, which reduces synthesis time and run-time memory usage.

If you are using a third-party synthesis tool, you can create separate atom netlist files for parts of your design that you already have synthesized and optimized so that you update only the parts of the design that change.

Regardless of your synthesis tool, you can use full incremental compilation along with LogicLock regions to preserve your placement and routing results for unchanged partitions while working on other partitions. This ability provides the most reduction in compilation time and run-time memory usage because neither synthesis nor fitting is performed for unchanged partitions in the design.

You can also perform a bottom-up compilation in which parts of the design are compiled completely independently in separate Quartus II projects, and then exported into the top-level design. This flow is useful in team-based designs or when incorporating third-party IP.



For information about the full incremental compilation flow in the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For information about using the Quartus II incremental synthesis feature alone, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about creating multiple netlist files in third-party tools for use with incremental compilation, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Use Multiple Processors for Parallel Compilation

The Quartus II software can run some algorithms in parallel to take advantage of multiple processors and reduce compilation time when more than one processor is available to compile the design. You can specify the maximum number of processors that the software can use. The Quartus II software supports up to four processors. The software does not necessarily use all the processors that you specify during a given compilation, but it never uses more than the specified number of processors. This allows you to work on other tasks on your machine without becoming slow or less responsive.

By allowing the Quartus II software to use two processors, you may be able to reduce the compilation time by up to 10% on systems with dual-core processors. Four processors can reduce compilation time by up to 15%. With certain design flows in which timing analysis runs alone, using multiple processors can reduce the time required for timing analysis by an average of 12% when using two processors. This reduction can reach an average of 15% when using four processors.

The actual reduction in compilation time depends on the design and on the specific settings used for compilation. For example, compilations with fast-corner optimization turned on benefit more from using multiple processors than do compilations that do not use fast-corner optimization. The runtime requirement is not reduced for some other compilation stages, such as Analysis and Synthesis. The Fitter (quartus_fit), the Classic Timing Analyzer (quartus_tan) and the TimeQuest Timing Analyzer (quartus_sta) stages in the compilation may benefit from the use of multiple processors. In the Compilation Report, on the **Flow Elapsed Time** panel, the average number of processors used for these stages is shown.



Do not consider processors with Intel Hyper-Threading to be more than one processor. If you have a single processor with Intel Hyper-Threading enabled, you should set the number of processors to one. Altera recommends that you do not use the Intel Hyper-Threading feature for Quartus II compilations, as it can increase runtimes.



Many factors can impact the performance of parallel compilation. For detailed information and instructions that can help improve the performance of this feature, refer to the solution to the problem "How can I improve the compilation time performance of the parallel compilation feature in the Quartus II software?" on the Altera website, **www.altera.com**.

The Quartus II software does not check the number of processors physically available, but instead simply configures its algorithms to use the specified number of processors. Therefore, you should not specify more processors than those actually available. Doing so could result in increased compilation time.

Using multiple processors does not affect the quality of the fit. For a given Fitter seed on a specific design, the fit is exactly the same, regardless of whether the Quartus II software uses one processor or multiple processors. The only difference between such compilations using different number of processors is the compilation time.

To set the number of processors available for Quartus II compilation, on the Assignments menu, select **Settings**. From the **Settings** dialog box, under Category, click **Compilation Process Settings**. In the dialog box that appears, specify the **Maximum processors for parallel Quartus II use**. The default value for the number of processors is 1.

You can also set the number of processors available for Quartus II compilation using the following Tcl command in your script.

```
set_global_assignment -name NUM_PARALLEL_PROCESSORS <value> ;
```

In this case, <*value*> is an integer between `1` and `4`.

Reduce Synthesis Time and Synthesis Netlist Optimization Time

You can reduce synthesis time by reducing your use of netlist optimizations and by using incremental compilation. Use incremental compilation (with Netlist Type set to Post-Synthesis) to reduce the synthesis time, without affecting the Fitter time. For more ideas about reducing synthesis time in third-party EDA synthesis tools, refer to your synthesis software's documentation.

Synthesis Netlist Optimizations

You can use Quartus II integrated synthesis to synthesize and optimize HDL designs, and you can use synthesis netlist optimizations to optimize netlists that were synthesized by third-party EDA software. Using these netlist optimizations can cause the Analysis and Synthesis module to take much longer to run. Look at the Analysis and Synthesis messages to find out how much time these optimizations take. The compilation time spent in Analysis and Synthesis is typically small compared to the compilation time spent in the Fitter.

If your design meets your performance requirements without synthesis netlist optimizations, turn off the optimizations to save time. If you need to turn on synthesis netlist optimizations to meet performance, you can optimize parts of your design hierarchy separately to reduce the overall time spent in analysis and synthesis.

Check Early Timing Estimation before Fitting

The Quartus II software can provide an estimate of your timing results after synthesis, before the design is fully processed by the Fitter. In cases where you want a quick estimate of your design results before proceeding with further design or synthesis tasks, this feature can save you significant compilation time. For more information, refer to "Early Timing Estimation" on page 8–9.

After you perform analysis and synthesis in the Quartus II software, in the Processing menu, point to **Start,** and click **Start Early Timing Estimate**.

Reduce Placement Time

The time needed to place a design depends on two factors: the number of ways the logic in the design can be placed in the device and the settings that control how hard the placer works to find a good placement. You can reduce the placement time in two ways:

- Change the settings for the placement algorithm.
- Use incremental compilation to preserve the placement for parts of the design.

Sometimes there is a trade-off between placement time and routing time. Routing time can increase if the placer does not run long enough to find a good placement. When you reduce placement time, make sure that it does not increase routing time and negate the overall time reduction.

Fitter Effort Setting

Standard Fit takes the most runtime and usually does not yield a better result than Auto Fit. To switch from Standard to Auto Fit, on the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**, and use the **Fitter effort** setting to shorten runtime by changing the effort level to **Auto Fit**. If you are certain that your design has only easy-to-meet timing constraints and low routing resource usage, you can select **Fast Fit** for an even greater runtime saving.

Placement Effort Multiplier Settings

You can control the amount of time the Fitter spends in placement by reducing one aspect of placement effort with the **Placement Effort Multiplier** option. On the Assignments menu, click **Settings**. Select Fitter Settings, and click **More Settings**. Under **Existing Option Settings**, select **Placement Effort Multiplier**. The default is `1.0`. Legal values must be greater than 0 and can be non-integer values. Numbers between 0 and 1 can reduce fitting time, but also can reduce placement quality and design performance. Numbers higher than 1 increase placement time and placement quality, but may reduce routing time for designs with routing congestion. For example, a value of 4 increases placement time by approximately 2 to 4 times, but may increase quality.

Final Placement Optimization Levels

The **Final Placement Optimization Level** option specifies whether the Fitter performs final placement optimizations. This can be set to **Always**, **Never**, and **Automatically**. Performing optimizations may improve register-to-register timing and fitting, but may require longer compilation times. The default setting of **Automatically** can be used with the **Auto Fit Fitter** Effort Level (also the default) to let the Fitter decide whether these optimizations should run based on the routability and timing requirements of the design.

Setting the Final Placement Optimization to **Never** often reduces your compilation time, but typically affects routablity negatively and reduces timing performance.

To change the Final Placement Optimization level, on the Assignments menu, choose **Settings**. The **Settings** dialog box appears. From the Category list, select **Fitter Settings**. Click the **More Settings** button. Select **Final Placement Optimization Level**, and then from the drop-down menu, select the required setting.

Physical Synthesis Effort Settings

You can use the physical synthesis options to optimize your post-synthesis netlist and improve your timing performance. These options, which affect placement, can significantly increase compilation time. Refer to Table 8–6 on page 8–57 for detailed results.

If your design meets your performance requirements without physical synthesis options, turn them off to save time. You also can use the **Physical synthesis effort** setting on the **Physical Synthesis Optimizations** page under **Fitter Settings** in the **Category** list to reduce the amount of extra compilation time that these optimizations use. The **Fast** setting directs the Quartus II software to use a lower level of physical synthesis optimization that, compared to the normal level, can cause a smaller increase in compilation time. However, the lower level of optimization can result in a smaller increase in design performance.

Limit to One Fitting Attempt

This option causes the software to quit after one fitting attempt option, instead of repeating placement and routing with increased effort.

From the Assignments menu, select **Settings**. On the Fitter Settings page, turn on **Limit to one fitting attempt**.

For more details about this option, refer to "Limit to One Fitting Attempt" on page 8–12.

Preserving Placement, Incremental Compilation, and LogicLock Regions

Preserving information about previous placements can make future placements take less time. The incremental compilation provides an easy-to-use methodology for preserving placement results. For more information, refer to "Incremental Compilation" on page 8–86 and the references listed in the section.

Reduce Routing Time

The time needed to route a design depends on three factors: the device architecture, the placement of the design in the device, and the connectivity between different parts of the design. Typically, the routing time is not a significant amount of the compilation time. If your design takes a long time to route, perform one or more of the following actions:

- Check for routing congestion
- Let the placer run longer to find a more routable placement
- Use incremental compilation to preserve routing information for parts of your design

Identify Routing Congestion in the Chip Planner

To identify areas of congested routing in your design, open the Chip Planner. On the Tools menu, click Chip Planner. To view the routing congestion in the Chip Planner, click the Layers icon located next to the Task menu. Under Background Color Map, select the Routing Utilization. Routing resource usage above 90% indicates routing congestion. You can change the connections in your design to reduce routing congestion. If the area with routing congestion is in a LogicLock region or between LogicLock regions, change or remove the LogicLock regions and recompile the design. If the routing time remains the same, the time is a characteristic of the design and the placement. If the routing time decreases, consider changing the size, location, or contents of LogicLock regions to reduce congestion and decrease routing time.



For information about identifying areas of congested routing using the Chip Planner tool, refer to the *Viewing Routing Congestion* subsection in the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Identify Routing Congestion in the Timing Closure Floorplan for Legacy Devices

If the device you have in your design is not supported by the Chip Planner, you have to use the Timing Closure Floorplan tool. To identify areas of congested routing in your design. To open Timing Closure Floorplan, on the Assignments menu, click **Timing Closure Floorplan,** and turn on **Show Routing Congestion**. This feature is available only when you choose the **Field View** on the View menu. Routing resource usage above 90% indicates routing congestion. You can change the connections in your design to reduce routing congestion. If the area with routing congestion is in a LogicLock region or between LogicLock regions, change or remove the LogicLock regions and recompile the design. If the routing time remains the same, the time is a characteristic of

the design and the placement. If the routing time decreases, consider changing the size, location, or contents of LogicLock regions to reduce congestion and decrease routing time.

Placement Effort Multiplier Setting

Some designs may be difficult to route, and take a long time to route because the placement is less than optimal. In such cases, you can increase the Placement Effort Multiplier to get a better placement. Though this may increase the placement time, it can reduce the routing time, and even overall compilation time in some cases.

Preserve Routing Incremental Compilation and LogicLock Regions

Preserving information about the previous routing results for part of the design can make future routing efforts take less time. The use of LogicLock regions with incremental compilation provides an easy-to-use methodology that preserves placement and routing results. For more information, refer to "Incremental Compilation" on page 8–86 and the references listed in the section.

Other Optimizing Resources

The Quartus II software has additional resources to help you optimize your design for resource, performance, compilation time and power.

Design Space Explorer

The Design Space Explorer (DSE) automates the process of running multiple compilations with different settings. You can use the DSE to try the techniques described in this chapter. The DSE utility helps automate the process of finding the best set of options for your design. The DSE explores the design space by applying various optimization techniques and analyzing the results.



For more information, refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*.

Power Optimization Advisor

The Quartus II software has a Power Optimization Advisor to provide guidance for reducing power consumption. In addition, the Incremental Compilation Advisor provides suggestions to improve your quality of results when partitioning your design for a hierarchical or team-based design flow using the Quartus II incremental compilation feature.



For more information about using the Power Optimization Advisor, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*. Fore more information about using the Incremental Compilation Advisor, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF form.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

You can specify many of the options described in this section either in an instance, or at a global level, or both.

Use the following Tcl command to make a global assignment:

```
set_global_assignment -name <QSF variable name> <value>
```

Use the following Tcl command to make an instance assignment:

```
set_instance_assignment -name <QSF variable name> <value> \
-to <instance name>
```



If the <*value*> field includes spaces (for example, "Standard Fit"), the value must be enclosed by straight double quotation marks.

Initial Compilation Settings

The Quartus Settings File variable name is used in the Tcl assignment to make the setting along with the appropriate value. The Type column indicates whether the setting is supported as a global setting, an instance setting, or both.



This chapter refers to timing settings and analysis in the Quartus II Classic Timing Analyzer. For equivalent settings and analysis in the Quartus II TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* or the *Switching to the Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*.

Table 8–13 lists the Quartus Settings File variable name and applicable values for the settings discussed in "Initial Compilation: Required Settings" on page 8–3.

Table 8–13. Initial Compilation Settings

Setting Name	Quartus Settings File Variable Name	Values	Type
Device Setting	DEVICE	*<device part number>*	Global
Use Smart Compilation	SPEED_DISK_USAGE_TRADEOFF	SMART, NORMAL	Global
Optimize IOC Register Placement For Timing	OPTIMIZE_IOC_REGISTER_PLACEMENT_FOR_TIMING	ON, OFF	Global
Optimize Hold Timing	OPTIMIZE_HOLD_TIMING	OFF, IO PATHS AND MINIMUM TPD PATHS, ALL PATHS	Global
Fitter Effort	FITTER_EFFORT	STANDARD FIT, FAST FIT, AUTO FIT	Global
Router Effort Multiplier	ROUTER_EFFORT_MULTIPLIER	Any positive, non-zero value	Global
Router Timing Optimization level	ROUTER_TIMING_OPTIMIZATION_LEVEL	NORMAL, MINIMUM, MAXIMUM	Global
Final Placement Optimization	FINAL_PLACEMENT_OPTIMIZATION	ALWAYS, AUTOMATICALLY, NEVER	Global

Resource Utilization Optimization Techniques (LUT-Based Devices)

Table 8–14 lists the Quartus Settings File variable name and applicable values for the settings discussed in "Resource Utilization Optimization Techniques (LUT-Based Devices)" on page 8–25. The QSF variable name is used in the Tcl assignment to make the setting along with the appropriate value. The Type column indicates whether the setting is supported as a global setting, an instance setting, or both.

Table 8–14. Resource Utilization Optimization Settings

Setting Name	QSF Variable Name	Values	Type
Auto Packed Registers *(1)*	AUTO_PACKED_REGISTERS_ *<device family name>*	OFF, NORMAL, MINIMIZE AREA, MINIMIZE AREA WITH CHAINS, AUTO	Global, Instance
Perform WYSIWYG Primitive Resynthesis	ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP	ON, OFF	Global, Instance
Optimization Technique	*<device family name>*_OPTIMIZATION_ TECHNIQUE	AREA, SPEED, BALANCED	Global, Instance
Speed Optimization Technique for Clock Domains	SYNTH_CRITICAL_CLOCK	ON, OFF	Instance
State Machine Encoding	STATE_MACHINE_PROCESSING	AUTO, ONE-HOT, MINIMAL BITS, USER-ENCODE	Global, Instance
Preserve Hierarchy	PRESERVE_HIERARCHICAL_BOUNDARY	OFF, RELAXED, FIRM	Instance
Auto RAM Replacement	AUTO_RAM_RECOGNITION	ON, OFF	Global, Instance
Auto ROM Replacement	AUTO_ROM_RECOGNITION	ON, OFF	Global, Instance
Auto Shift Register Replacement	AUTO_SHIFT_REGISTER_RECOGNITION	ON, OFF	Global, Instance
Auto Block Replacement	AUTO_DSP_RECOGNITION	ON, OFF	Global, Instance
Number of Processors for Parallel Compilation	NUM_PARALLEL_PROCESSORS	Integer between 1 and 4 inclusive	Global

Note to Table 8–15:
(1) Allowed values for this setting depend on the device family that is selected.

I/O Timing Optimization Techniques (LUT-Based Devices)

Table 8–15 lists the QSF variable name and applicable values for the settings discussed in "I/O Timing Optimization Techniques (LUT-Based Devices)" on page 8–97. The QSF variable name is used in the Tcl assignment to make the setting along with the appropriate value. The **Type** column indicates whether the setting is supported as a global setting, an instance setting, or both.

Table 8–15. I/O Timing Optimization Settings

Setting Name	Quartus Settings File Variable Name	Values	Type
Optimize IOC Register Placement For Timing	OPTIMIZE_IOC_REGISTER_PLACEMENT_FOR_TIMING	ON, OFF	Global
Fast Input Register	FAST_INPUT_REGISTER	ON, OFF	Instance
Fast Output Register	FAST_OUTPUT_REGISTER	ON, OFF	Instance
Fast Output Enable Register	FAST_OUTPUT_ENABLE_REGISTER	ON, OFF	Instance
Fast OCT Register	FAST_OCT_REGISTER	ON, OFF	Instance

Register-to-Register Timing Optimization Techniques (LUT-Based Devices)

Table 8–16 lists the QSF variable name and applicable values for the settings discussed in "Register-to-Register Timing Optimization Techniques (LUT-Based Devices)" on page 8–52. The QSF variable name is used in the Tcl assignment to make the setting along with the appropriate value. The **Type** column indicates whether the setting is supported as a global setting, an instance setting, or both.

Table 8–16. Register-to-Register Timing Optimization Settings (Part 1 of 2)

Setting Name	Quartus Settings File Variable Name	Values	Type
Perform WYSIWYG Primitive Resynthesis	ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP	ON, OFF	Global, Instance
Perform Gate Level Register Retiming	ADV_NETLIST_OPT_SYNTH_GATE_RETIME	ON, OFF	Global
Allow Register Retiming to trade off t_{SU}/t_{CO} with f_{MAX}	ADV_NETLIST_OPT_RETIME_CORE_AND_IO	ON, OFF	Global

Table 8–16. Register-to-Register Timing Optimization Settings (Part 2 of 2)

Setting Name	Quartus Settings File Variable Name	Values	Type
Perform Physical Synthesis for Combinational Logic	PHYSICAL_SYNTHESIS_COMBO_LOGIC	ON, OFF	Global
Perform Register Duplication	PHYSICAL_SYNTHESIS_REGISTER_DUPLICATION	ON, OFF	Global
Perform Register Retiming	PHYSICAL_SYNTHESIS_REGISTER_RETIMING	ON, OFF	Global
Physical Synthesis Effort	PHYSICAL_SYNTHESIS_EFFORT	NORMAL, EXTRA, FAST	Global
Fitter Seed	SEED	*<integer>*	Global
Maximum Fan-Out	MAX_FANOUT	*<integer>*	Instance
Manual Logic Duplication	DUPLICATE_ATOM	*<node name>*	Instance
Optimize Power during Synthesis	OPTIMIZE_POWER_DURING_SYNTHESIS	NORMAL, OFF EXTRA_EFFORT	Global
Optimize Power during Fitting	OPTIMIZE_POWER_DURING_FITTING	NORMAL, OFF EXTRA_EFFORT	Global

Duplicate Logic for Fan-Out Control

The manual logic duplication option accepts wildcards. This is an easy and powerful duplication technique that you can use without editing your source code. You can use this technique, for example, to make a duplicate of a large fan-out node for all of its destinations in a certain design hierarchy, such as `hierarchy_A`. To make such an assignment with Tcl, use a command similar to Example 8–1.

Example 8–1. Duplication Technique

```
set_instance_assignment -name DUPLICATE_ATOM \
  high_fanout_to_A -from high_fanout_node \
  -to *hierarchy_A*
```

Conclusion

Complex designs have complex requirements. Methodologies for fitting your design and for achieving timing closure are fundamental to optimal performance of your design. Using the Quartus II design optimization methodology closes timing quickly on complex designs, reduces iterations by providing more intelligent and better linkage between analysis and assignment tools, and balances multiple design constraints including multiple clocks, routing resources, and area constraints.

The Quartus II software provides many features to achieve optimal results. Follow the techniques presented in this chapter to efficiently optimize a design for area or timing performance, or to reduce compilation time.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Analyzing Designs with Quartus II Netlist Viewers* chapter in volume 1 of the *Quartus II Handbook*
- *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*
- *Design Analysis and Engineering Change Management with Chip Planner* chapter in volume 3 of the *Quartus II Handbook*
- *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Instantiating Altera Megafunctions* and the *Inferring Altera Megafunctions* sections of the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Message Suppression* section in the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 8–17 shows the revision history for this chapter.

Table 8–17. Document Revision History (Part 1 of 3)

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Major update for the Quartus II software version 7.2, including: • Major re-organization of the chapter	Changes made to this chapter reflect the software changes made in version 7.2.
May 2007 v7.1.0	Updated for the Quartus II software version 7.1, including: • Adding text to “Limit to One Fitting Attempt” on page 8–12 • Significantly adjusted text to “Fitter Effort Setting” on page 8–13 • Added Table 8–1, “Typical Register Packing Results for Arria GX Devices,” on page 8-30 • Adjusted text throughout to include Chip Planner information • Adjusted text throughout to add Arria GX support information • Added a paragraph on page 8–89 for more information about identifying routing congestion in the timing closure floorplan • Added row to Table 8–15, “Resource Utilization Optimization Settings,” on page 8-94 • Added “Referenced Documents” on page 8–97	Changes made to this chapter reflect the software changes made in version 7.1.
March 2007 v7.0.0	Minor changes to add information regarding Cyclone III details, including updating benchmarking tables: • Table 8–2, “Typical Register Packing Results for Stratix II and Stratix III Devices,” on page 8-30 • Table 8–3, “Typical Register Packing Results for Cyclone II and Cyclone III Devices,” on page 8-30	Added Cyclone III information.

Table 8–17. Document Revision History (Part 2 of 3)

Date and Document Version	Changes Made	Summary of Changes
November 2006 v6.1.0	Updated for the Quartus II software version 6.1.0: ● Added references to the Power Optimization Advisor and Incremental Compilation Advisor ● Updated text to include Quartus II TimeQuest Timing Analyzer ● Added `check_timing` for illegal and ignored constraints ● In the "Resource Utilization" section, modified note about ALM counts ● In the "Use Register Packing" section, updated for Stratix II support; added new benchmarking tables ● Added new sections to the "Routing" section: • Set Auto Register Packing to Auto • Set Fitter Aggressive Routability Optimizations to Always • Increase Router Effort Multiplier • Set Maximum Router Optimization Level ● To the "Increase Placement Effort Multiplier" section, added that second and third fitting loops increase the multiplier to 4 and then 16 ● Added relevant information for Stratix III ● In the "Synthesis Netlist Optimizations and Physical Synthesis Optimizations" section, updated benchmarking information in tables; reorganized information for clarity ● Added new section: "Turn Off Extra-Effort Power Optimization Settings"	Updates for TimeQuest support, Stratix III devices, and updated benchmarking tables.

Table 8–17. Document Revision History (Part 3 of 3)

Date and Document Version	Changes Made	Summary of Changes
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0: ● Added Optimization advisors. ● Added initial compilation information. ● Added design analysis information. ● Added f_{MAX} timing optimization techniques.	—
December 2005 v5.1.1	Minor typographic corrections.	—
October 2005 v5.1.0	Chapter 8 was formerly Chapter 7 in version 5.0.	—
May 2005 v5.0.0	Chapter 7 was formerly Chapter 6 in version 4.2.	—
Dec. 2004 v2.1	Updated for Quartus II software version 4.2: ● Re-organized chapter. ● Added Early Timing Estimation segment. ● Removed Incremental Fitting segment. ● Updated Optimization Advisors. ● Updated Resource Utilization Optimization Techniques (LUT-Based Devices) segment. ● Added the DSP Block Balancing logic option to Retarget or Balance DSP Blocks segment. ● Updated Duplicate Logic for Fan-Out Control segment. ● Updates to tables, figures.	—
June 2004 v2.0	● Updates to tables, figures. ● New functionality in the Quartus II software version 4.1.	—
Feb. 2004 v1.0	Initial release.	—

9. Power Optimization

QII52016-7.2.0

Introduction

The Quartus® II software offers power-driven compilation to fully optimize device power consumption. Power-driven compilation focuses on reducing your design's total power consumption using power-driven synthesis and power-driven place-and-route. This chapter describes the power-driven compilation feature and flow in detail, as well as low power design techniques that can further reduce power consumption in your design. The techniques primarily target Arria™ GX, Stratix® and Cyclone® series of devices, and HardCopy® II devices. These devices utilize a low-k dielectric material that dramatically reduces dynamic power and improves performance. Stratix III and Stratix II devices include new, more efficient, logic structures called adaptive logic modules (ALMs) that obtain maximum performance while minimizing power consumption. Cyclone III and Cyclone II devices offer the optimal blend of high performance and low power in a low-cost FPGA.



For more information about Stratix III architecture, refer to the *Stratix III Device Handbook*.

Altera provides the Quartus II PowerPlay Power Analyzer to aid you during the design process by delivering fast and accurate estimations of power consumption. You can minimize power consumption, while taking advantage of the industry's leading FPGA performance, by using the tools and techniques described in this chapter.



For more information about the PowerPlay Power Analyzer, refer to the *PowerPlay Power Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Total FPGA power consumption is comprised of I/O power, core static power, and core dynamic power. This chapter focuses on design optimization options and techniques that help reduce core dynamic power and I/O power. In addition to these techniques there are additional power optimization techniques available for Stratix III devices. These techniques include:

- Selectable Core Voltage
- Programmable Power Technology
- Device Speed Grade Selection



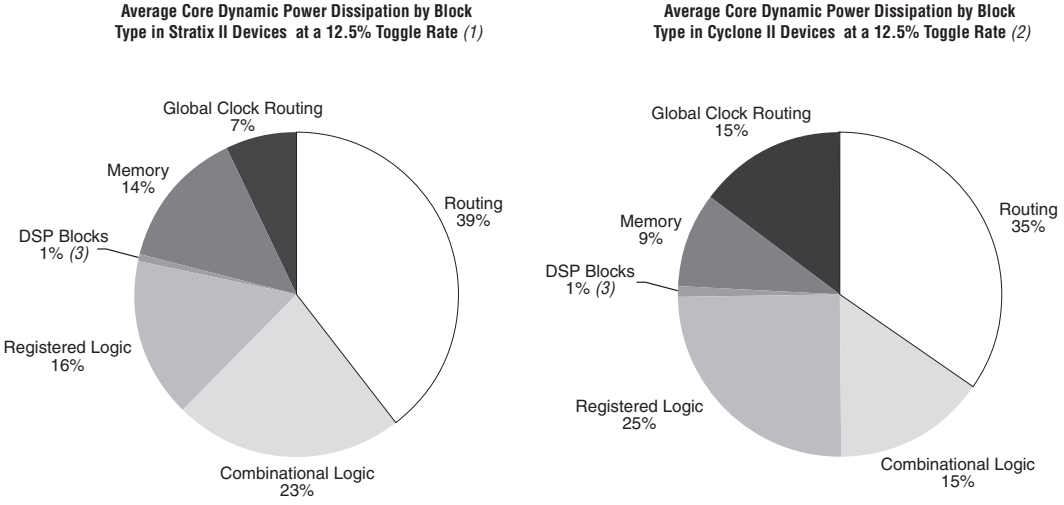
For more information about power optimization techniques available for Stratix III devices, refer to application note *AN: 437 Power Optimization in Stratix III FPGAs*.

Power Dissipation

This section describes the sources of power dissipation in Stratix II and Cyclone II devices. You can refine techniques that reduce power consumption in your design by understanding the sources of power dissipation.

Figure 9–1 shows the power dissipation of Stratix II and Cyclone II devices in different designs. All designs were analyzed at a fixed clock rate of 200 MHz and exhibited varied logic resource utilization across available resources.

Figure 9–1. Average Core Dynamic Power Dissipation



Notes to Figure 9–1:
(1) 112 different designs were used to obtain these results.
(2) 93 different designs were used to obtain these results.
(3) In designs using DSP blocks, DSPs consumed 5% of core dynamic power.

As shown in Figure 9–1, a significant amount of the total power is dissipated in routing for both Stratix II and Cyclone II devices, with the remaining power dissipated in logic, clock, and RAM blocks.

In Stratix and Cyclone device families, a series of column and row interconnect wires of varying lengths provide signal interconnections between logic array blocks (LABs), memory block structures, and digital signal processing (DSP) blocks or multiplier blocks. These interconnects dissipate the largest component of device power.

FPGA combinational logic is another source of power consumption. The basic building block of logic in Stratix III and Stratix II devices is the ALM, and in Cyclone III and Cyclone II devices, it is the logic element (LE).



For more information about ALMs and LEs in Stratix III, Stratix II, Cyclone III and Cyclone II devices, refer to the respective device handbook.

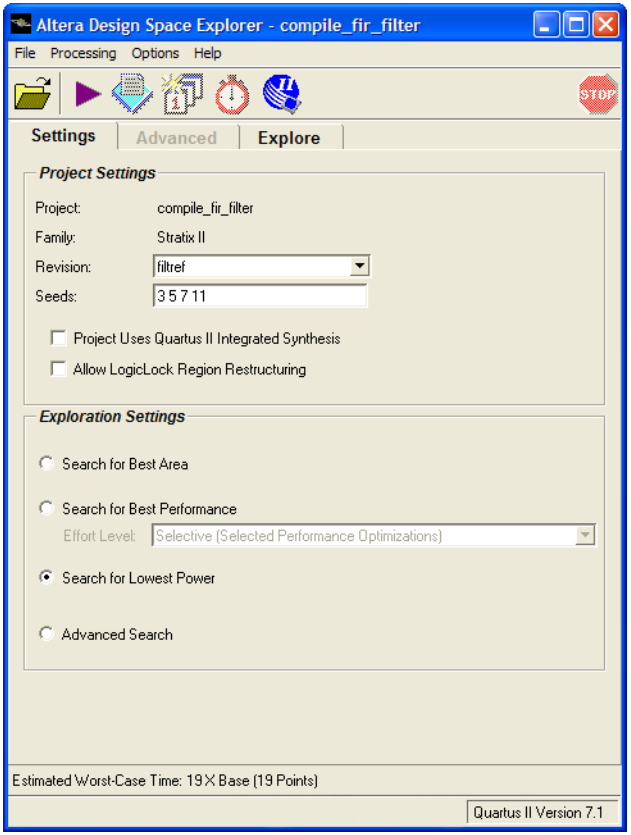
Memory and clock resources are other major consumers of power in FPGAs. Stratix II devices feature the TriMatrix memory architecture. TriMatrix memory includes 512-bit M512 blocks, 4-Kbit M4K blocks, and 512-Kbit M-RAM blocks, which are each configurable to support many features. Stratix III TriMatrix on-chip memory is an enhancement based upon the Stratix II FPGA TriMatrix memory and includes three sizes of memory blocks: MLAB blocks, M9K blocks, and M144K blocks. Cyclone II devices have 4-Kbit M4K memory blocks and Cyclone III devices have 9-Kbit M9K memory blocks.

Design Space Explorer

The Design Space Explorer (DSE) is a simple, easy-to-use, design optimization utility that is included in the Quartus II software. The DSE explores and reports optimal Quartus II software options for your design, targeting either power optimization, design performance, or area utilization improvements. You can use the DSE to implement the techniques described in this chapter.

Figure 9–2 shows the DSE user interface. The **Settings** tab is divided into **Project Settings** and **Exploration Settings**.

Figure 9–2. Design Space Explorer User Interface



The **Search for Lowest Power** option, under **Exploration Settings**, uses a predefined exploration space that targets overall design power improvements. This setting focuses on applying different options that specifically reduce total design thermal power. You can also set the **Optimization Goal** option for your design using the **Advanced** tab in the DSE window. You can select your design optimization goal, such as optimize for power, from the list of available settings in the **Optimization Goal** list. The DSE then uses the selection from the **Optimization Goal** list, along with the **Search for Lowest Power** selection, to determine the best compilation results.

By default, the Quartus II PowerPlay Power Analyzer is run for every exploration performed by the DSE when the **Search for Lowest Power** option is selected. This helps you debug your design and determine trade-offs between power requirements and performance optimization.



For more information about the DSE, refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*.

Power-Driven Compilation

The standard Quartus II compilation flow consists of Analysis and Synthesis, Fitter, Assembler, and Timing Analysis. Power-driven compilation takes place at the analysis and synthesis and Fitter levels. Power-driven compilation settings are divided in the **PowerPlay power optimization** list on the **Analysis & Synthesis Settings** page, and **PowerPlay power optimization** on the **Fitter Settings** page. The following section describes these power optimization options at the analysis and synthesis and Fitter levels.

Power-Driven Synthesis

Synthesis netlist optimization occurs during the synthesis stage of the compilation flow. The optimization technique makes changes to the synthesis netlist to optimize your design according to the selection of area, speed, or power optimization. This section describes power optimization techniques at the synthesis level.

The **Analysis & Synthesis Settings** page allows you to specify logic synthesis options. The **PowerPlay power optimization** option is available for the Arria GX, Stratix and Cyclone families of devices, and MAX® II devices (Figure 9–3).

To perform power optimization at the synthesis level in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis**. The **Analysis & Synthesis** page appears.
3. In the **PowerPlay power optimization** list, select your preferred setting. This option determines how aggressively Analysis and Synthesis optimizes the design for power.

Figure 9–3. Analysis & Synthesis Settings Page

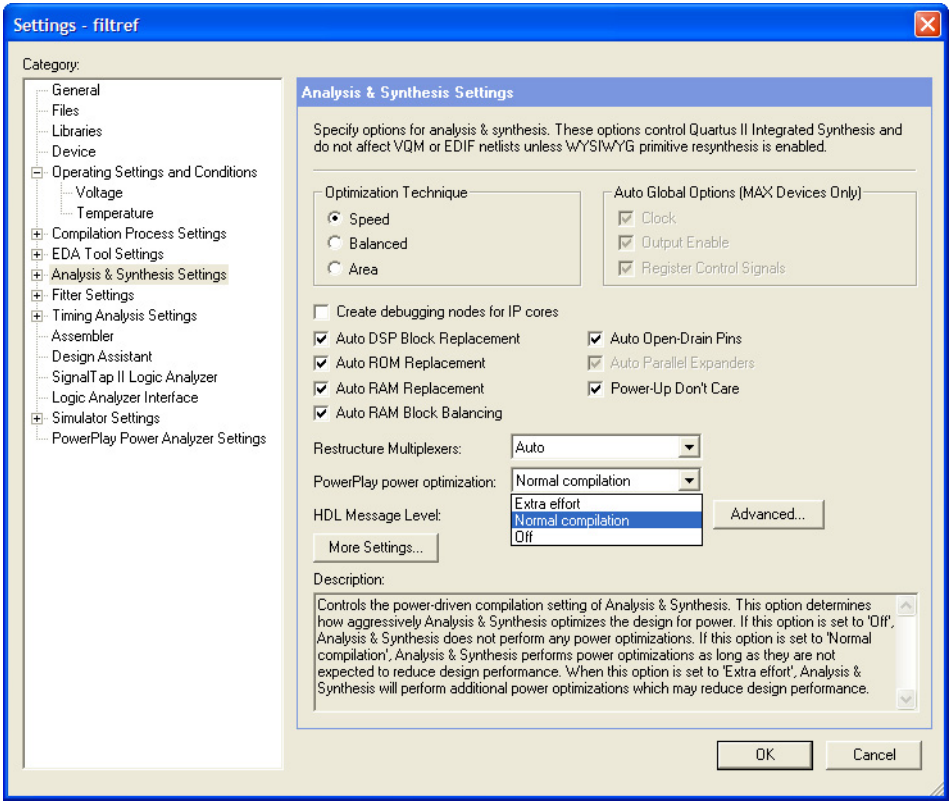


Table 9–1 shows the settings in the **PowerPlay power optimization** list. You can apply these settings on a project or entity level.

Table 9–1. Optimize Power During Synthesis Options

Settings	Description
Off	No netlist, placement, or routing optimizations are performed to minimize power
Normal compilation (Default)	Enables power optimizations as long as they are not expected to reduce design performance
Extra effort	Enables you to perform additional power optimizations which can reduce design performance

The **Normal compilation** setting is turned on by default. This setting performs memory optimization and power-aware logic mapping during synthesis.

Memory blocks can represent a large fraction of total design dynamic power as described in "Reducing Memory Power Consumption" on page 9–23. Minimizing the number of memory blocks accessed during each clock cycle can significantly reduce memory power. Memory optimization involves effective movement of user-defined read/write enable signals to associated read-and-write clock enable signals for all memory types (Figure 9–4).

Figure 9–4. Memory Transformation

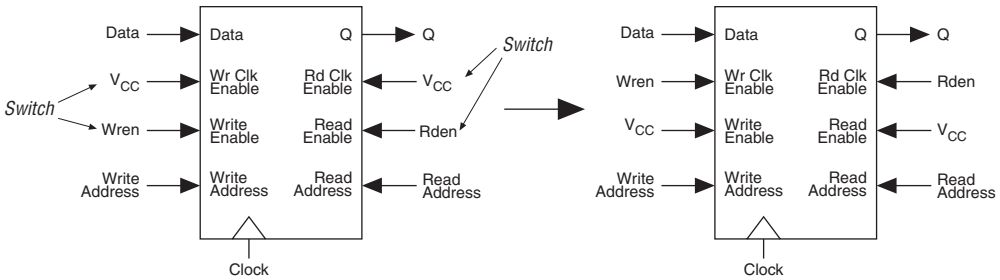


Figure 9–4 shows a default implementation of a simple dual-port memory block in which write-clock enable and read-clock enable signals are connected to V_{CC}, making both read-and-write memory ports active during each clock cycle. Memory transformation effectively moves the read-enable and write-enable signals to the respective read-clock enable and write-clock enable signals. By using this technique, memory ports are shut down when they are not accessed. This significantly reduces your design's memory power consumption. For more information about clock enable signals, refer to "Reducing Memory Power Consumption" on page 9–23. For Stratix III devices, the memory transformation takes place at the Fitter level by selecting the **Normal compilation** settings for the power optimization option.



In Stratix III and Cyclone III devices, the specified read-during-write behavior can significantly impact the power of single-port and bidirectional dual-port RAMs. It is best to set the read-during-write parameter to "Don't care" (at the HDL level), as it allows an optimization whereby the read-enable signal can be set to the inversion of the existing write-enable signal (if one exists). This allows the core of the RAM to shut down (that is, not toggle), which saves a significant amount of power.

The other type of power optimization that takes place with the **Normal compilation** setting is power-aware logic mapping. The power-aware logic mapping reduces power by rearranging the logic during synthesis to eliminate nets with high toggle rates.

The **Extra effort** setting performs the functions of the **Normal compilation** setting and other memory optimizations to further reduce memory power by shutting down memory blocks that are not accessed. This level of memory optimization may require extra logic which can reduce design performance.

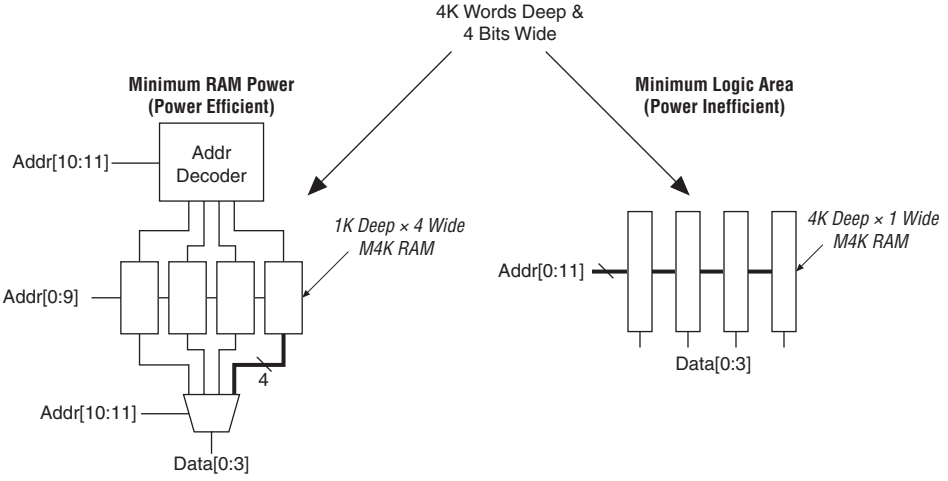
The **Extra effort** setting also performs power-aware memory balancing. Power-aware memory balancing automatically chooses the best memory configuration for your memory implementation and provides optimal power saving by determining the number of memory blocks, decoder, and multiplexer circuits needed. If you have not previously specified target-embedded memory blocks for your design's memory functions, the power-aware balancer automatically selects them during memory implementation.

Figure 9–5 shows an example of a 4K × 4 (4K deep and 4 bits wide) memory implementation in two different configurations using M4K memory blocks available in Stratix II devices. The minimum logic area implementation uses M4K blocks configured as 4K × 1. This implementation is the default in the Quartus II software because it has the minimum logic area (0 logic cells) and the highest speed. However, all four M4K blocks are active on each memory access in this implementation, which increases RAM power. The minimum RAM power implementation is created by selecting **Extra effort** in the **PowerPlay power optimization** list. This implementation automatically uses four M4K blocks configured as 1K × 4 for optimal power saving. An address decoder is implemented by the altsyncram megafunction to select which of the four M4K blocks should be activated on a given cycle, based on the state of the top two user address bits. The altsyncram megafunction automatically implements a multiplexer to feed the downstream logic by choosing the appropriate M4K output. This

implementation reduces RAM power because only one M4K block is active on any cycle, but it requires extra logic cells, costing logic area and potentially impacting design performance.

There is a trade-off between power saved by accessing fewer memories and power consumed by the extra decoder and multiplexor logic. The Quartus II software automatically balances the power savings against the costs to choose the lowest power configuration for each logical RAM.

Figure 9–5. 4K × 4 Memory Implementation Using Multiple M4K Blocks



Memory optimization options can also be controlled by the `Low_Power_Mode` parameter in the **Default Parameters** page of the **Settings** dialog box. The settings for this parameter are **None**, **Auto**, and **ALL**. **None** corresponds to the **Off** setting in the **PowerPlay power optimization** list. **Auto** corresponds to the **Normal compilation** setting and **ALL** corresponds to the **Extra effort** setting, respectively. You can apply PowerPlay power optimization either on a compiler basis or on individual entities. The `Low_Power_Mode` parameter always takes precedence over the Optimize Power for Synthesis option for power optimization on memory.

You can also set the `MAXIMUM_DEPTH` parameter manually to configure the memory for low power optimization. This technique is the same as the power-aware memory balancer, but it is manual rather than automatic like the **Extra effort** setting in the **PowerPlay power optimization** list. You can set the `MAXIMUM_DEPTH` parameter for memory modules manually in the megafunction instantiation or in the MegaWizard® Plug-In Manager for power optimization as described in

"Reducing Memory Power Consumption" on page 9–23. The MAXIMUM_DEPTH parameter always takes precedence over the Optimize Power for Synthesis options for power optimization on memory optimization.

Power-Driven Synthesis Experiment for Stratix II Devices

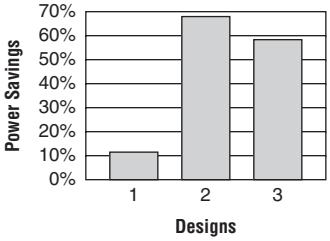
In this experiment for Stratix II devices, three designs are compiled with the Quartus II software using **Normal compilation** and **Extra effort** settings in the **PowerPlay power optimization** list. The default setting for Fitter is **Normal compilation**. Table 9–2 shows resources used in the power-driven synthesis experiment.

Table 9–2. Resources Used in the Power-Driven Synthesis Experiment for Stratix II Devices

Design Name	Settings	ALUT	Register	Memory Bits
Design 1	Normal compilation	8,941	9,150	293,856
	Extra effort	8,954	9,151	293,856
Design 2	Normal compilation	28,169	12,148	1,009,920
	Extra effort	28,817	12,297	1,009,920
Design 3	Normal compilation	5,376	2,809	153,864
	Extra effort	5,559	2,813	153,864

The PowerPlay Power Analyzer estimates the power using a gate-level simulation output file. Figure 9–6 shows that the power-driven synthesis reduces memory power consumption by as much as 68% in Stratix II devices.

Figure 9–6. Memory Blocks Power Savings Using the Power-Driven Synthesis for Stratix II Devices



Power-Driven Fitter

The **Fitter Settings** page enables you to specify options for fitting (Figure 9–7). The **PowerPlay power optimization** option is available for Arria GX, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II devices.

To perform power optimization at the Fitter level, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Fitter Settings**. The **Fitter Settings** page appears.
3. In the **PowerPlay power optimization** list, select your preferred setting. This option determines how aggressively the Fitter optimizes the design for power.

Figure 9–7. Fitter Settings Page

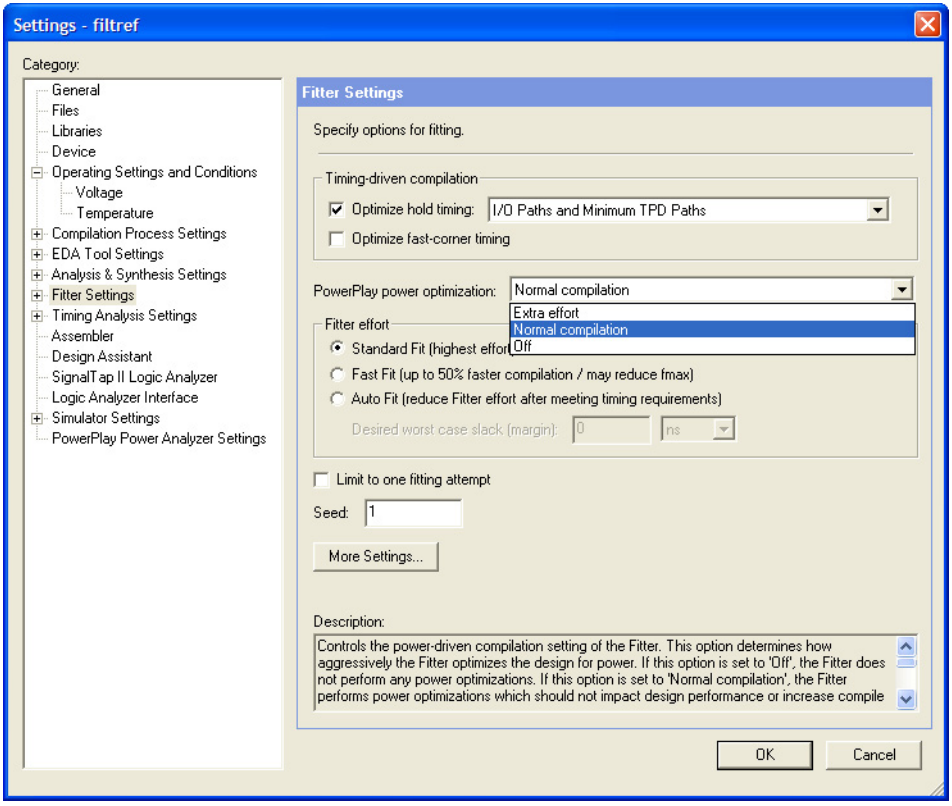


Table 9–3 lists the settings in the **PowerPlay power optimization** list. These settings can only be applied on a project-wide basis. The **Extra effort** setting for the Fitter requires extensive effort to optimize the design for power and can increase the compilation time.

Table 9–3. Power-Driven Fitter Option

Settings	Description
Off	No netlist, placement, or routing optimizations are performed to minimize power
Normal compilation (Default)	Enables power optimizations as long as they are not expected to reduce design performance
Extra effort	Enables you to perform additional power optimizations that can reduce design performance

The **Normal compilation** setting is selected by default and performs DSP optimization by creating power-efficient DSP block configurations for your DSP functions. For Stratix III devices, this setting, which is based on timing constraints entered for the design, enables the Programmable Power Technology to configure tiles as high-speed mode or low-power mode. Programmable Power Technology is always turned **ON** even when the **OFF** setting is selected for the Fitter PowerPlay power optimization option. Tiles are the combination of LAB and MLAB pairs (including the adjacent routing associated with LAB and MLAB) which can be configured to operate in high-speed or low-power mode. This level of power optimization will not have any affect on the fitting, timing results, or compile time. Also, for Stratix III devices, this setting enables the memory transformation as described in "Power-Driven Synthesis" on page 9–5.



For more information about Stratix III power optimization, refer to *AN 437:Power Optimization in Stratix III FPGAs*.

The **Extra effort** setting performs the functions of the **Normal compilation** setting and other place-and-route optimizations during fitting to fully optimize the design for power. The Fitter applies an extra effort to minimize power even after timing requirements have been met by effectively moving the logic closer during placement to localize high-toggling nets, and using routes with low capacitance. However, this effort can increase the compilation time.

The **Extra effort** setting uses a Signal Activity File (**.saf**) or Verilog Value Change Dump File (**.vcd**) that guides the Fitter to fully optimize the design for power based on the signal activity of the design. The best power optimization during fitting results from using the most accurate signal activity information. Signal activities from full post-fit netlist (timing) simulation provide the highest accuracy because all node activities reflect the actual design behavior, provided that supplied input vectors are representative of typical design operation. If you do not have a Signal Activity File (from simulation or other source), the Quartus II software uses assignments, clock assignments, and vectorless estimation values (PowerPlay Power Analyzer Tool settings) to estimate the signal activities. This information is used to optimize your design for power during fitting.



Only the **Extra effort** setting in the **PowerPlay power optimization** list for the Fitter option uses the signal activities (from Value Change Dump File or SAF) during fitting. The settings made in the **PowerPlay Power Analyzer Settings** page in the **Settings** dialog box are used to calculate the signal activity of your design.

For more information about Signal Activity Files and Verilog Value Change Dump Files, and how to create them, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Power-Driven Fitter Experiment for Stratix II Devices

In this experiment for Stratix II devices, three designs are compiled with the Quartus II software using the **Normal compilation** and **Extra effort** settings in the Fitter for the PowerPlay power optimization list. The default setting for Analysis and Synthesis is **Normal compilation**.

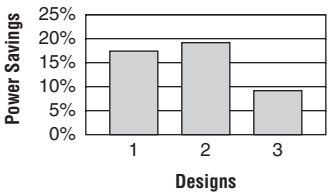
Table 9–4 shows resources used in the power-driven Fitter experiment.

Table 9–4. Designs Used in the Power-Driven Fitter Experiment for Stratix II Devices

Design Name	ALUTs (Normal Compilation)	ALUTs (Extra Effort)
Design 1	21,435	21,363
Design 2	19,035	18,970
Design 3	5,335	5,328

The PowerPlay Power Analyzer estimates the power using a gate-level simulation output file. Figure 9–8 shows that the power-driven Fitter technique reduces power consumption by as much as 19% in Stratix II devices.

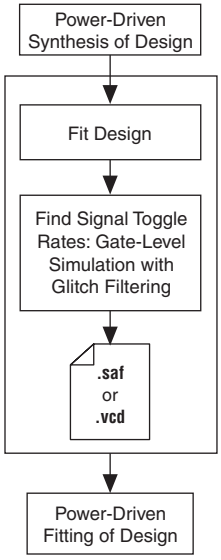
Figure 9–8. Power Savings Using the Power-Driven Fitter for Stratix II Devices



Recommended Flow for Power-Driven Compilation

Figure 9–9 shows the recommended design flow to fully optimize your design for power during compilation. This flow utilizes the power-driven synthesis and power-driven Fitter options. On average, you can reduce core dynamic power by 16% with the extra effort synthesis and extra effort fitting settings, as compared to off settings in both synthesis and Fitter options for power-driven compilation.

Figure 9–9. Recommended Flow for Power-Driven Compilation



Area-Driven Synthesis

Using area optimization rather than timing or delay optimization during synthesis saves power because you use fewer logic blocks. Using less logic usually means less switching activity.

Area-Driven Synthesis Experiment for Stratix II Devices

In this experiment for Stratix II devices, five designs are compiled with the Quartus II software in two ways. First, the designs are compiled optimizing for area. The same designs are then compiled optimizing for speed. The power optimization settings for synthesis and fitting are set to **Off**.

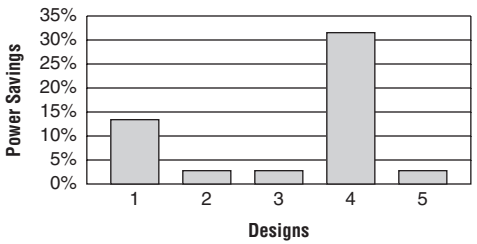
Table 9–5 shows ALUT usage in the area-driven synthesis experiment.

Table 9–5. Designs Used in the Area-Driven Synthesis Experiment for Stratix II Devices

Design Name	ALUTs (Area Mapping)	ALUTs (Speed Mapping)
Design 1	5,682	8,553
Design 2	16,986	17,783
Design 3	36,554	36,312
Design 4	4,717	5,820
Design 5	15,947	15,978

The PowerPlay Power Analyzer estimates the power using a gate-level simulation output file. Figure 9–10 shows that the area-driven technique reduces power consumption by as much as 31% in Stratix II devices.

Figure 9–10. Power Savings Using Area-Driven Synthesis for Stratix II Devices



Area-Driven Synthesis Experiment for Cyclone II Devices

In this experiment for Cyclone II devices, five designs are compiled with the Quartus II software in two ways. First, the designs are compiled optimizing for area. The same designs are then compiled optimizing for speed.

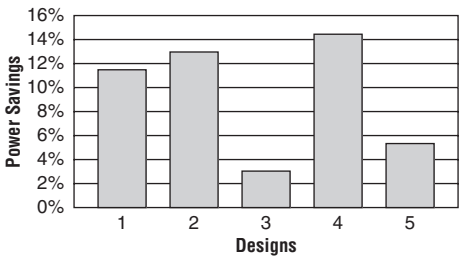
Table 9–6 shows LE usage in the area-driven synthesis experiment.

Table 9–6. Designs Used in the Area-Driven Synthesis Experiment for Cyclone II Devices

Design Name	LEs (Area Mapping)	LEs (Speed Mapping)
Design 1	13,020	16,429
Design 2	13,317	13,636
Design 3	5,384	5,690
Design 4	33,640	40,008
Design 5	21,409	22,988

The PowerPlay Power Analyzer estimates the power using a gate-level simulation output file. Figure 9–11 shows that the area-driven technique reduces power consumption by as much as 15% in Cyclone II devices.

Figure 9–11. Power Savings Using Area-Driven Synthesis for Cyclone III and Cyclone II Devices

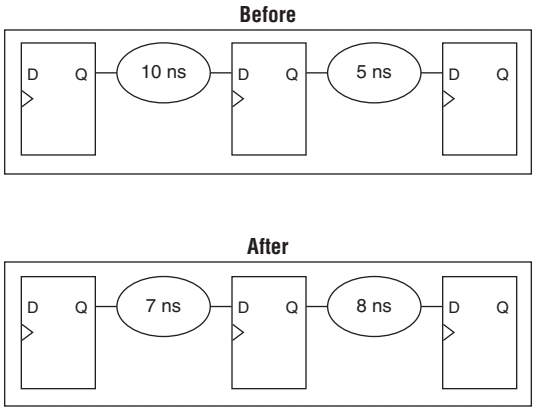


Gate-Level Register Retiming

You can also use gate-level register retiming to reduce circuit switching activity. Retiming shuffles registers across combinational blocks without changing design functionality. The **Perform gate-level register retiming** option in the Quartus II software enables the movement of registers across combinational logic to balance timing, allowing the software to trade off the delay between timing critical and non-critical timing paths.

Retiming uses fewer registers than pipelining. Figure 9–12 shows an example of gate-level register retiming, where the 10 ns critical delay is reduced by moving the register relative to the combinational logic, resulting in the reduction of data depth and switching activity.

Figure 9–12. Gate-Level Register Retiming



Gate-level register retiming makes changes at the gate level. If you are using an atom netlist from a third-party synthesis tool, you must also select the **Perform WYSIWYG primitive resynthesis** option to undo the atom primitives to gates mapping (so that register retiming can be performed), and then to remap gates to Altera® primitives. When using the Quartus II integrated synthesis, retiming occurs during synthesis before the design is mapped to Altera primitives.

For more information about register retiming, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

Gate-Level Register Retiming Experiment for Stratix II Devices

In this experiment for Stratix II devices, three designs are compiled with the Quartus II software in two ways. First, a netlist from a third-party synthesis tool is compiled. Then, the same netlist is compiled after selecting the **Perform WYSIWYG primitive resynthesis** and **Perform gate-level register retiming** options.

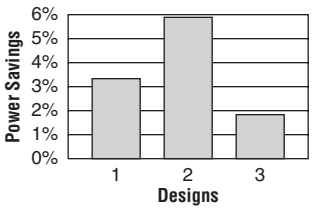
Table 9–7 shows resource usage results.

Table 9–7. Resources Used in the Gate-Level Register Retiming Experiment for Stratix II Devices

Design Name	WYSIWYG and Register Retiming	ALUTs	Registers	DSP Blocks	Memory
Design 1	No	2,051	691	0	16
	Yes	1,882	731	0	16
Design 2	No	123,909	40,070	0	0
	Yes	95,593	39,816	0	0
Design 3	No	6,354	6,019	64	3,584
	Yes	7,496	5,970	64	3,584

The PowerPlay Power Analyzer estimates the power using a gate-level simulation output file. Figure 9–13 shows that the combination of WYSIWYG remapping and gate-level register retiming reduces power consumption by nearly 6% in Stratix II devices.

Figure 9–13. Power Savings Using Retiming for Stratix II Devices



Gate-Level Register Retiming Experiment for Cyclone II Devices

In this experiment for Cyclone II devices, three designs are compiled with the Quartus II software in two ways. First, a netlist from a third-party synthesis tool is compiled. Then, the same netlist is compiled by selecting the **Perform WYSIWYG primitive resynthesis** and **Perform gate-level register retiming** options.

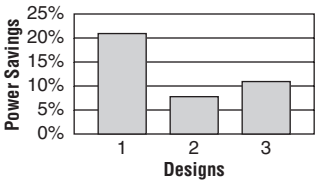
Table 9–8 shows resource usage results.

Table 9–8. Resources Used in the Gate-Level Register Retiming Experiment for Cyclone II Devices

Design Name	WYSIWYG and Register Retiming	LEs	Registers	Multiplier Blocks	Memory
Design 1	No	385	137	0	0
	Yes	278	143	0	0
Design 2	No	14,758	1,683	0	0
	Yes	13,079	1,683	0	0
Design 3	No	31,727	29,097	96	3,120
	Yes	27,038	24,272	96	3,120

The PowerPlay Power Analyzer estimates the power using a gate-level simulation output file. Figure 9–14 shows that the combination of WYSIWYG remapping and gate-level register retiming reduces power consumption by as much as 21% in Cyclone II devices.

Figure 9–14. Power Savings Using Retiming for Cyclone II Devices



Design Guidelines

Several low-power design techniques can reduce power consumption when applied during FPGA design implementation. This section provides detailed design techniques for Stratix III, Stratix II, Cyclone III, and Cyclone II devices that affect overall design power. The results of these techniques may be different from design to design.

Clock Power Management

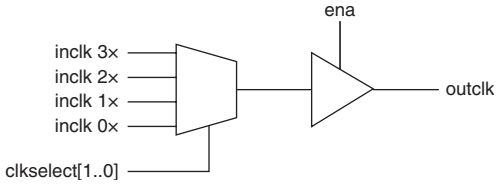
Clocks represent a significant portion of dynamic power consumption due to their high switching activity and long paths. Figure 9–1 on page 9–2 shows a 7% average contribution to power consumption for global clock routing in Stratix II devices and 15% in Cyclone II devices. Actual clock-related power consumption is higher than this because the

power consumed by local clock distribution within logic, memory, and DSP or multiplier blocks is included in the power consumption for the respective blocks.

Clock routing power is automatically optimized by the Quartus II software, which only enables those portions of the clock network that are required to feed downstream registers. Power can be further reduced by gating clocks when they are not needed. It is possible to build clock gating logic, but this approach is not recommended because it is difficult to generate a glitch-free clock in FPGAs using ALMs or LEs.

Arria GX, Stratix III, Stratix II, Cyclone III, and Cyclone II devices use clock control blocks that include an enable signal. A clock control block is a clock buffer that lets you dynamically enable or disable the clock network and dynamically switch between multiple sources to drive the clock network. You can use the Quartus II MegaWizard Plug-In Manager to create this clock control block with the altclkctrl megafunction. Arria GX, Stratix III, Stratix II, Cyclone III, and Cyclone II devices provide clock control blocks for global clock networks. In addition, Stratix III and Stratix II devices have clock control blocks for regional clock networks. The dynamic clock enable feature lets internal logic control the clock network. When a clock network is powered down, all the logic fed by that clock network does not toggle, thereby reducing the overall power consumption of the device. Figure 9–15 shows a 4-input clock control block diagram.

Figure 9–15. Clock Control Block Diagram

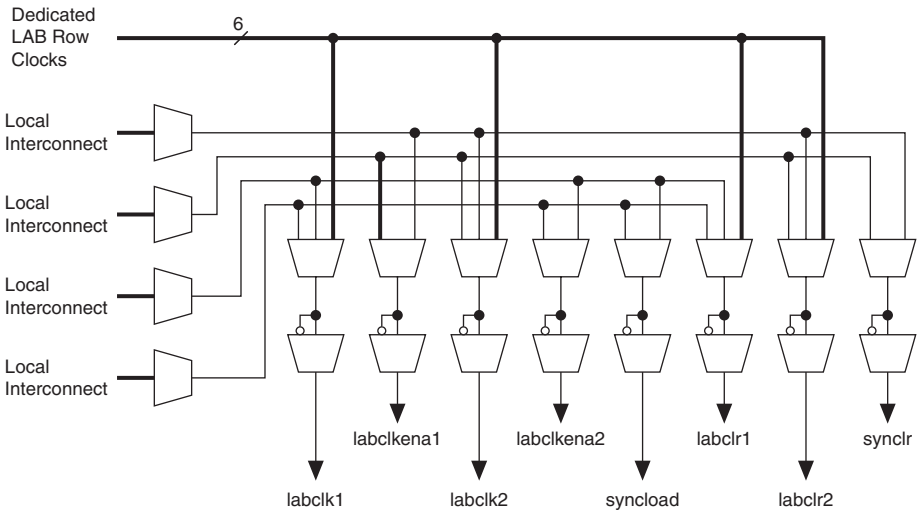


The enable signal is applied to the clock signal before being distributed to global routing. Therefore, the enable signal can either have a significant timing slack (at least as large as the global routing delay) or it can reduce the f_{MAX} of the clock signal.

For more information about using clock control blocks, refer to the *altclkctrl Megafunction User Guide*.

Another contributor to clock power consumption is the LAB clock that distributes a clock to the registers within a LAB. LAB clock power can be the dominant contributor to overall clock power. For example, in Cyclone III and Cyclone II devices, each LAB can use two clocks and two clock enable signals, as shown in Figure 9–16. Each LAB's clock signal and clock enable signal are linked. For example, an LE in a particular LAB using the `labclk1` signal also uses the `labclkena1` signal.

Figure 9–16. LAB-Wide Control Signals



To reduce LAB-wide clock power consumption without disabling the entire clock tree, use the LAB-wide clock enable to gate the LAB-wide clock. The Quartus II software automatically promotes register-level clock enable signals to the LAB-level. All registers within an LAB that share a common clock and clock enable are controlled by a shared gated clock. To take advantage of these clock enables, use a clock enable construct in the relevant HDL code for the registered logic.

LAB-Wide Clock Enable Example

This VHDL code makes use of a LAB-wide clock enable. This clock-gating logic is automatically turned into an LAB-level clock enable signal.

```
IF clk'event AND clock = '1' THEN
    IF logic_is_enabled = '1' THEN
        reg <= value;
    ELSE
```

```
            reg <= reg;
        END IF;
    END IF;
```

For more information about LAB-wide control signals, refer to the *Stratix II Architecture*, *Cyclone III Architecture*, or *Cyclone II Architecture* chapters in the respective device handbook.

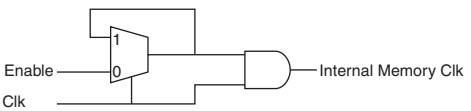
Reducing Memory Power Consumption

The memory blocks in FPGA devices can represent a large fraction of typical core dynamic power. Memory represents 14% of the core dynamic power in a typical Stratix II device design and 9% in a Cyclone II device design. Memory blocks are unlike most other blocks in the device because most of their power is tied to the clock rate, and is insensitive to the toggle rate on the data and address lines.

When a memory block is clocked, there is a sequence of timed events that occur within the block to execute a read or write. The circuitry controlled by the clock consumes the same amount of power regardless of whether or not the address or data has changed from one cycle to the next. Thus, the toggle rate of input data and the address bus have no impact on memory power consumption.

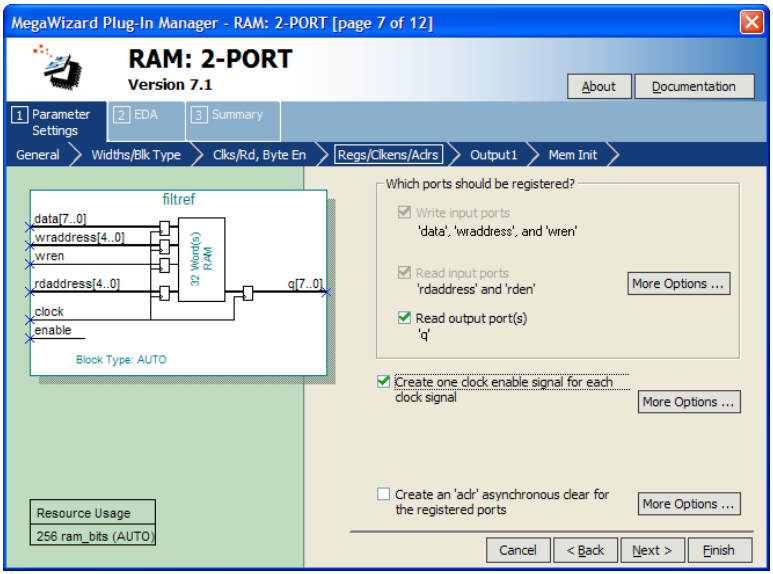
The key to reducing memory power consumption is to reduce the number of memory clocking events. You can achieve this through clock network-wide gating described in "Clock Power Management" on page 9–20, or on a per-memory basis through use of the clock enable signals on the memory ports. Figure 9–17 shows the logical view of the internal clock of the memory block. Use the appropriate enable signals on the memory to make use of the clock enable signal instead of gating the clock.

Figure 9–17. Memory Clock Enable Signal



Using the clock enable signal enables the memory only when necessary and shuts it down for the rest of the time, reducing the overall memory power consumption. You can use the Quartus II MegaWizard Plug-In Manager to create these enable signals by selecting the **Clock enable signal** option for the appropriate port when generating the memory block function (Figure 9–18).

Figure 9–18. MegaWizard Plug-In Manager RAM 2-Port Clock Enable Signal Selectable Option

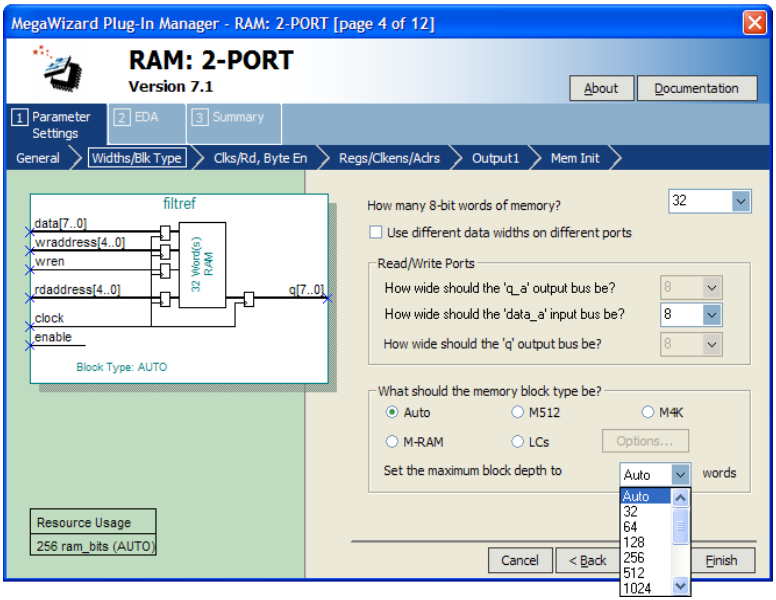


For example, consider a design that contains a 32-bit-wide M4K memory block in ROM mode that is running at 200 MHz. Assuming that the output of this block is only needed approximately every four cycles, this memory block will consume 8.45 mW of dynamic power according to the demands of the downstream logic. By adding a small amount of control logic to generate a read clock enable signal for the memory block only on the relevant cycles, the power can be cut 75% to 2.15 mW.

You can also use the `MAXIMUM_DEPTH` parameter in your memory megafunction to save power in Stratix III, Stratix II, Cyclone III, and Cyclone II devices; however, this approach may increase the number of LEs required to implement the memory and affect design performance.

You can set the `MAXIMUM_DEPTH` parameter for memory modules manually in the megafunction instantiation or in the MegaWizard Plug-In Manager (Figure 9–19). The Quartus II software automatically chooses the best design memory configuration for optimal power as, described in "Power-Driven Compilation" on page 9–5.

Figure 9–19. MegaWizard Plug-In Manager RAM 2-Port Maximum Depth Selectable Option



Memory Power Reduction Example

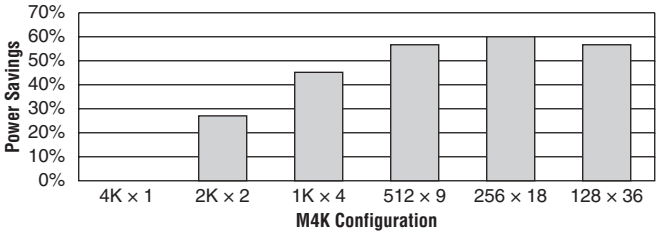
Table 9–9 shows power usage measurements for a 4K × 36 simple dual-port memory implemented using multiple M4K blocks in a Stratix II EP2S15 device. For each implementation, the M4K blocks are configured with a different memory depth.

Table 9–9. 4K × 36 Simple Dual-Port Memory Implemented Using Multiple M4K Blocks

M4K Configuration	Number of M4K Blocks	ALUTs
4K × 1 (Default setting)	36	0
2K × 2	36	40
1K × 4	36	62
512 × 9	32	143
256 × 18	32	302
128 × 36	32	633

Figure 9–20 shows the amount of power saved using the MAXIMUM_DEPTH parameter. For all implementations, a user-provided read enable signal is present to indicate when read data is needed. Using this power-saving technique can reduce power consumption by as much as 60%.

Figure 9–20. Power Savings Using the MAXIMUM_DEPTH Parameter



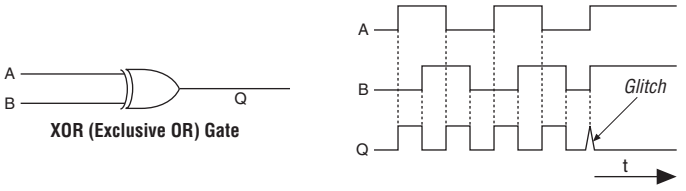
As the memory depth becomes more shallow, memory dynamic power decreases because unaddressed M4K blocks can be shut off using a decoded combination of address bits and the read enable signal. For a 128-deep memory block, power used by the extra LEs starts to outweigh the power gain achieved by using a more shallow memory block depth. The power consumption of the memory blocks and associated LEs depends on the memory configuration.

Pipelining and Retiming

Designs with many glitches consume more power because of faster switching activity. Glitches cause unnecessary and unpredictable temporary logic switches at the output of combinational logic. A glitch usually occurs when there is a mismatch in input signal timing leading to unequal propagation delay.

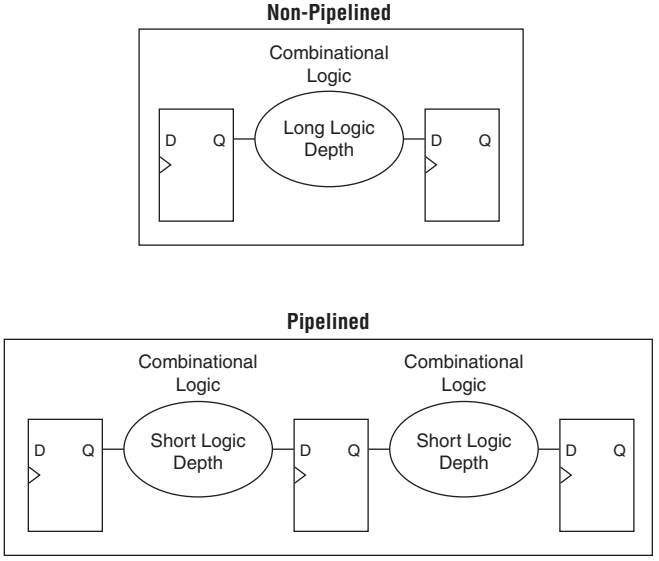
For example, consider an input change on one input of a 2-input XOR gate from 1 to 0, followed a few moments later by an input change from 0 to 1 on the other input. For a moment, both inputs become 1 (high) during the state transition, resulting in 0 (low) at the output of the XOR gate. Subsequently, when the second input transition takes place, the XOR gate output becomes 1 (high). During signal transition, a glitch is produced before the output becomes stable, as shown in Figure 9–21. This glitch can propagate to subsequent logic and create unnecessary switching activity, increasing power consumption. Circuits with many XOR functions, such as arithmetic circuits or cyclic redundancy check (CRC) circuits, tend to have many glitches if there are several levels of combinational logic between registers.

Figure 9–21. XOR Gate Showing Glitch at the Output



Pipelining can reduce design glitches by inserting flipflops into long combinational paths. Flip-flops do not allow glitches to propagate through combinational paths. Therefore, a pipelined circuit tends to have less glitching. Pipelining has the additional benefit of generally allowing higher clock speed operations, although it does increase the latency of a circuit (in terms of the number of clock cycles to a first result). Figure 9–22 shows an example where pipelining is applied to break up a long combinational path.

Figure 9–22. Pipelining Example



Pipelining is very effective for glitch-prone arithmetic systems because it reduces switching activity, resulting in reduced power dissipation in combinational logic. Additionally, pipelining allows higher-speed

operation by reducing logic-level numbers between registers. The disadvantage of this technique is that if there are not many glitches in your design, pipelining may increase power consumption by adding unnecessary registers. Pipelining can also increase resource utilization.

Pipelining Experiment for Stratix II Devices

In this experiment, three designs are implemented in Stratix II devices with and without pipelining. These three designs use arithmetic heavily (based on XOR functions) and may result in significant glitching.

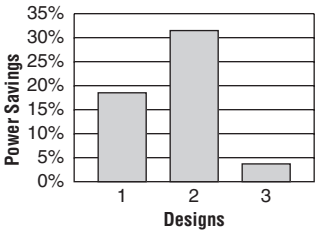
Table 9–10 shows the resource utilization for the designs used in the experiment.

Table 9–10. Resources Used in the Pipelining Experiment for Stratix II Devices

Design Name	Pipelined	ALUTs	Registers
Multiplier (Design 1)	No	9,726	448
	Yes	9,772	1,109
Accumulator multipliers (Design 2)	No	13,719	1,120
	Yes	14,007	2,260
Fir filter (Design 3)	Yes (level 1)	1,048	949
	Yes (level 2)	932	929

The PowerPlay Power Analyzer estimates the power using a gate-level simulation output file. Figure 9–23 shows that pipelining reduces dynamic power consumption by as much as 31% in Stratix II devices.

Figure 9–23. Power Savings Using Pipelining for Stratix II Devices



Pipelining Experiment for Cyclone II Devices

In this experiment, three designs are implemented in Cyclone II devices with and without pipelining. These three designs use arithmetic heavily (based on XOR functions) and may result in significant glitching.

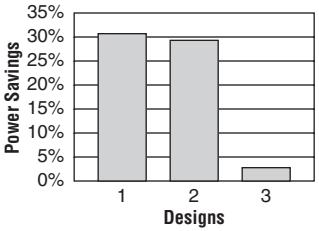
Table 9–11 shows resource utilization for the designs used in the experiment.

Table 9–11. Resources Used in the Pipelining Experiment for Cyclone II Devices

Design Name	Pipelined	LEs	Registers
Accumulator Multipliers (Design 1)	No	6,870	320
	Yes	13,071	3,719
Adder (Design 2)	No	7,392	1,076
	Yes	7,343	752
Divider (Design 3)	No	6,659	320
	Yes	6,735	520

The PowerPlay Power Analyzer estimates the power using a gate-level simulation output file. Figure 9–24 shows that pipelining reduces dynamic power by as much as 31% in Cyclone II devices.

Figure 9–24. Power Savings Using Pipelining for Cyclone II Devices



Architectural Optimization

You can use design-level architectural optimization by taking advantage of specific device architecture features. These features include dedicated memory and DSP or multiplier blocks available in FPGA devices to perform memory or arithmetic-related functions. You can use these

blocks in place of LUTs to reduce power consumption. For example, you can build large shift registers from RAM-based FIFO buffers instead of building the shift registers from the LE registers.

The Stratix device family allows you to efficiently target small, medium, and large memories with the TriMatrix memory architecture. Each TriMatrix memory block is optimized for a specific function. The M512 memory blocks available in Stratix II devices are useful for implementing small FIFO buffers, DSP, and clock domain transfer applications. M512 memory blocks are more power-efficient than the distributed memory structures in some competing FPGAs. The M4K memory blocks are used to implement buffers for a wide variety of applications, including processor code storage, large look-up table implementation, and large memory applications. The M-RAM blocks are useful in applications where a large volume of data must be stored on-chip. Effective utilization of these memory blocks can have a significant impact on power reduction in your design.

The Cyclone device family has configurable M4K and M9K memory blocks that provide various memory functions such as RAM, FIFO buffers, and ROM.



For more information about using DSP and memory blocks efficiently, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Architectural Optimization Experiment for Stratix II Devices

In this experiment, three designs are implemented in Stratix II devices in three ways to illustrate the power-reducing capabilities of dedicated blocks. The first two designs use logic elements and DSP blocks. The third design uses M4K and M-RAM blocks. In the third design, you can see that using MRAM blocks is more power efficient than using M4K blocks for large memory applications. The power optimization options for synthesis and fitting are turned off in this experiment.

Table 9–12 shows relative resource usage results.

Table 9–12. Designs Used in the Architectural Optimization Experiment for Stratix II Devices (Part 1 of 2)

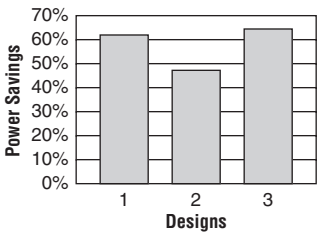
Design Name	Implementation	ALUT	Register	DSP Blocks	Memory
Design 1	Regular implementation	9,726	448	0	0
	Dedicated resource implementation	1,124	448	121	0

Table 9–12. Designs Used in the Architectural Optimization Experiment for Stratix II Devices (Part 2 of 2)

Design Name	Implementation	ALUT	Register	DSP Blocks	Memory
Design 2	Regular implementation	13,719	1,120	0	0
	Dedicated resource implementation	2,880	896	212	0
Design 3	M4K	286	228	0	1,835,008 (M4K)
	M-RAM	224	224	0	1,835,008 (M-RAM)

The PowerPlay Power Analyzer estimates the power using a gate-level simulation output file. Figure 9–25 shows that the architectural optimization technique has power savings of over 60% in Stratix II devices.

Figure 9–25. Power Savings Using Dedicated Blocks for Stratix II Devices



Architectural Optimization Experiment for Cyclone II

In this experiment, three designs are implemented in Cyclone II devices in three ways to illustrate the power-reducing capabilities of dedicated blocks. The first two designs use LEs and multiplier blocks. The third design uses LEs and M4K blocks.

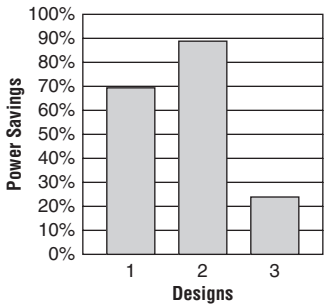
Table 9–13 shows relative resource usage results.

Table 9–13. Designs Used in the Architectural Optimization Experiment for Cyclone II Devices

Design Name	Implementation	LEs	Register	Multiplier Blocks	Memory
Design 1	Regular implementation	6,870	320	0	0
	Dedicated resource implementation	1,130	320	49	0
Design 2	Regular implementation	7,343	752	0	0
	Dedicated resource implementation	1,401	608	44	0
Design 3	Regular implementation	1,550	1,265	0	0
	M4K	72	72	0	1,152

The PowerPlay Power Analyzer estimates the power using a gate-level simulation output file. Figure 9–26 shows that the architectural optimization technique has power savings of as much as 88% in Cyclone II devices.

Figure 9–26. Power Savings Using Dedicated Blocks for Cyclone II Devices



I/O Power Guidelines

Non-terminated I/O standards such as LVTTL and LVCMOS have a rail-to-rail output swing. The voltage difference between logic-high and logic-low signals at the output pin is equal to the V_{CCIO} supply voltage. If the capacitive loading at the output pin is known, the dynamic power consumed in the I/O buffer can be calculated as:

$$P = 0.5 \times F \times C \times V^2$$

In this equation, *F* is the output transition frequency and *C* is the total load capacitance being switched. *V* is equal to V_{CCIO} supply voltage. Because of the quadratic dependence on V_{CCIO}, lower voltage standards consume significantly less dynamic power. In addition, lower pin capacitance is an important factor in considering I/O power consumption. Hardware and simulation data show that Stratix II device I/O pins have half the pin capacitance of the nearest competing FPGA. Cyclone II devices exhibit 20% less I/O power consumption than competitive, low-cost, 90 nm FPGAs.

Transistor-to-transistor logic (TTL) I/O buffers consume very little static power. As a result, the total power consumed by a LVTTL or LVCMOS output is highly dependent on load and switching frequency.

When using resistively terminated I/O standards like SSTL and HSTL, the output load voltage swings by a small amount around some bias point. The same dynamic power equation is used, where *V* is the actual load voltage swing. Because this is much smaller than V_{CCIO}, dynamic power is lower than for non-terminated I/O under similar conditions. These resistively terminated I/O standards dissipate significant static (frequency-independent) power, because the I/O buffer is constantly driving current into the resistive termination network. However, the lower dynamic power of these I/O standards means they often have lower total power than LVCMOS or LVTTL for high-frequency applications. Use the lowest drive strength I/O setting that meets your speed and waveform requirements to minimize I/O power when using resistively terminated standards.

You can save a small amount of static power by connecting unused I/O banks to the lowest possible V_{CCIO} voltage of 1.2 V.

Table 9–14 shows the total supply and thermal power consumed by outputs using different I/O standards for Stratix II devices. The numbers are for an I/O pin transmitting random data clocked at 200 MHz with a 10 pF capacitive load.

Table 9–14. I/O Power for Different I/O Standards in Stratix II Devices

Standard	Total Supply Current Drawn from V_{CCIO} Supply (mA)	Total On-Chip Thermal Power Dissipation (mW)
3.3-V LVTTL	2.42	9.87
2.5-V LVCMOS	1.9	6.69
1.8-V LVCMOS	1.34	4.18
1.5-V LVCMOS	1.18	3.58
3.3-V PCI	2.47	10.23
SSTL-2 class I	6.07	4.42
SSTL-2 class II	10.72	5.1
SSTL-18 class I	5.33	3.28
SSTL-18 class II	8.56	4.06
HSTL-15 class I	6.06	3.49
HSTL-15 class II	11.08	4.87
HSTL-18 class I	6.87	4.09
HSTL-18 class II	12.33	5.82

For this configuration, non-terminated standards generally use less power, but this is not always the case. If the frequency or the capacitive load is increased, the power consumed by non-terminated outputs increases faster than the power of terminated outputs.

For more information about I/O Standards, refer to the *Selectable I/O Standards in Stratix II Devices and Stratix II GX Devices* chapter in volume 2 of the *Stratix II Device Handbook* or the *Selectable I/O Standards in Cyclone II Devices* chapter in the *Cyclone II Device Handbook* or the *Cyclone III Device Handbook*.

When calculating I/O power, the PowerPlay Power Analyzer uses the default capacitive load set for the I/O standard in the **Capacitive Loading** tab of the **Device & Pin Options** dialog box. If **Enable Advanced I/O Timing** is turned on, I/O power is measured using an equivalent load calculated as the sum of the near capacitance, the transmission line distributed capacitance, and the far end capacitance as defined in the **Board Trace Model** tab of the **Device & Pin Options** dialog box or the

Board Trace Model view in the Pin Planner. Any other components defined in the board trace model are not taken into account for the power measurement.



For information about using Advanced I/O Timing and configuring a board trace model, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

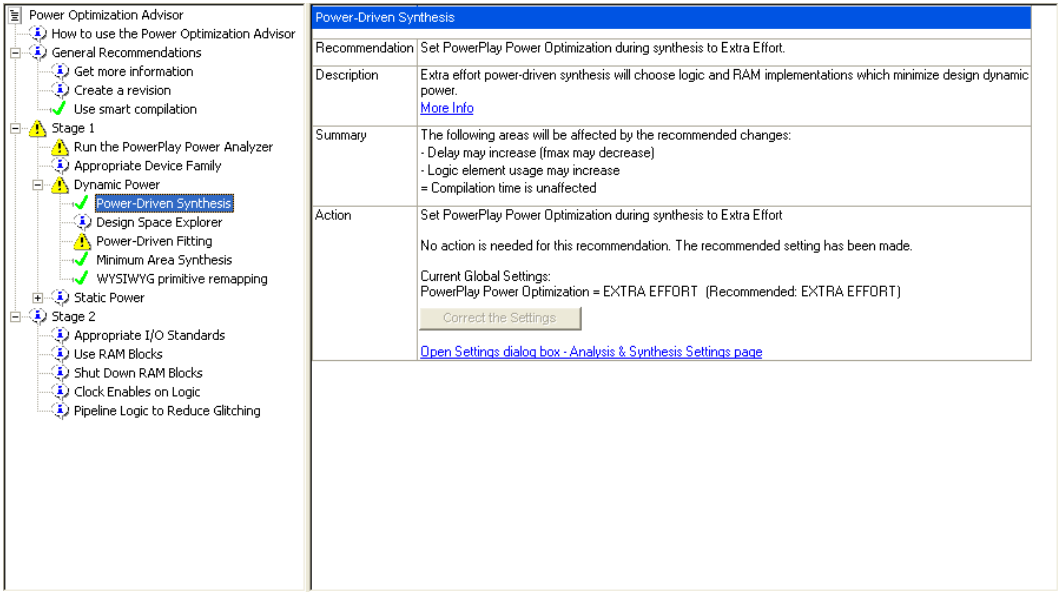
Power Optimization Advisor

The Quartus II software includes the Power Optimization Advisor, which provides specific power optimization advice and recommendations based on the current design project settings and assignments. The advisor covers many of the suggestions listed in this chapter. The following example shows how to reduce your design power with the Power Optimization Advisor.

Power Optimization Advisor Example

After compiling your design, run the PowerPlay Power Analyzer to determine your design power and to see where power is dissipated in your design. Based on this information, you can run the Power Optimization Advisor to implement recommendations that can reduce design power. Figure 9–27 shows the Power Optimization Advisor after compiling a design that is not fully optimized for power.

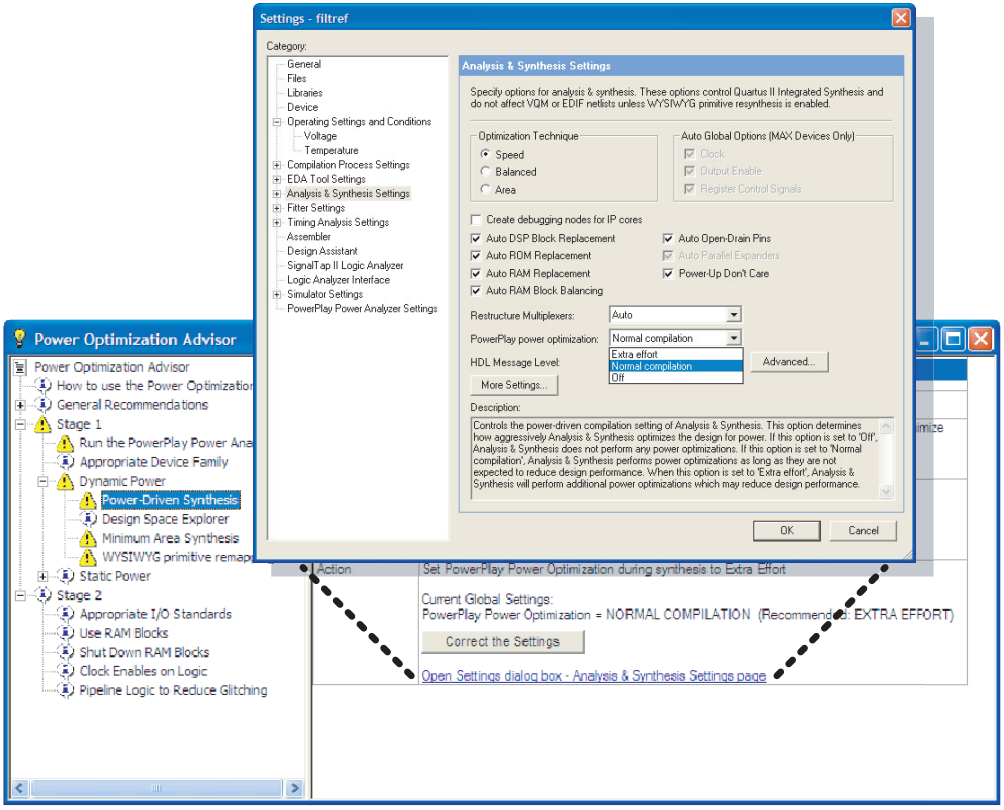
Figure 9–27. Power Optimization Advisor



The Power Optimization Advisor shows the recommendations that can reduce power in your design. The recommendations are split into stages to show the order in which you should apply the recommended settings. The first stage shows the mostly CAS settings options that are easy to implement and highly effective in reducing design power. An icon indicates whether each recommended setting is made in the current project. In Figure 9–27, the checkmark icon for Stage 1 shows the recommendations that are already implemented. The warning icons indicate recommendations that are not followed for this compilation. The information icon shows the general suggestions. Each recommendation includes the description, summary of the affect of the recommendation, and the action required to make the appropriate setting.

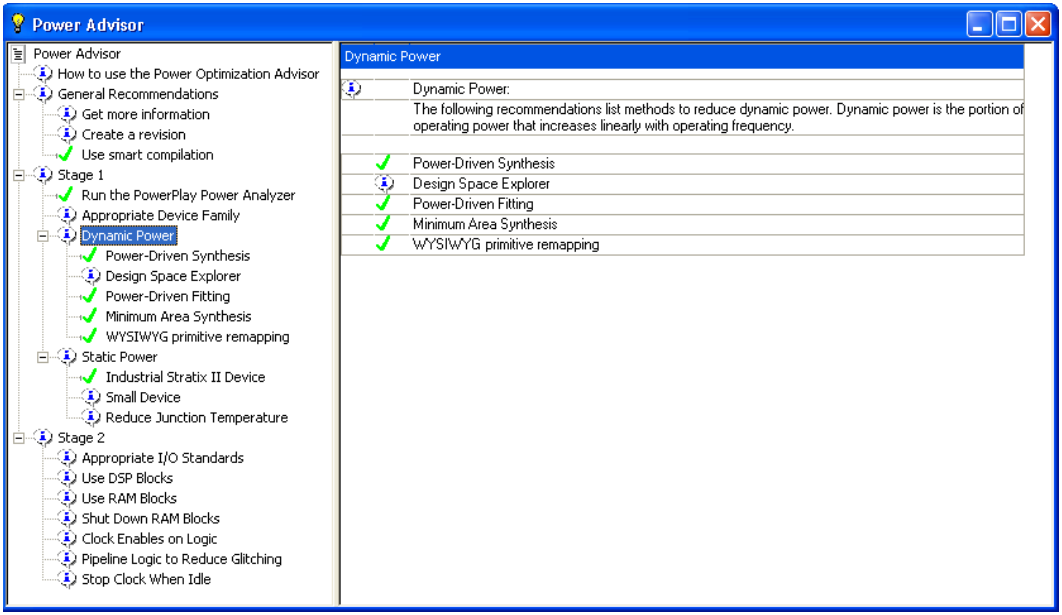
There is a link from each recommendation to the appropriate location in the Quartus II user interface where you can change the setting, such as the **Power-Driven Synthesis** setting. You can change the **Power-Driven Synthesis** setting by clicking **Open Settings dialog box - Analysis & Synthesis Settings page** (Figure 9–28). The **Settings** dialog box is shown with the **Analysis & Synthesis Settings** page selected, where you can change the **PowerPlay power optimization** settings.

Figure 9–28. Analysis & Synthesis Settings Page



After making the recommended changes, recompile your design. The Power Optimization Advisor indicates with green checkmarks that the recommendations were implemented successfully (Figure 9–29). You can use the PowerPlay Power Analyzer to verify your design power results.

Figure 9–29. Implementation of Power Optimization Advisor Recommendations



The recommendations listed in Stage 2 generally involve design changes, rather than CAD settings changes as in Stage 1. You can use these recommendations to further reduce your design power consumption. Altera recommends that you implement Stage 1 recommendations first, then the Stage 2 recommendations.

Conclusion

The combination of a smaller process technology, the use of low-k dielectric material, and reduced supply voltage significantly reduces dynamic power consumption in the latest FPGAs. To further reduce your dynamic power, you should use the design recommendations presented in this chapter to optimize resource utilization and minimize power consumption.

Referenced Documents

This chapter references the following documents:

- *altclkctrl Megafunction User Guide*
- *AN: 437 Power Optimization in Stratix III FPGAs*
- *Cyclone III Device Family Overview* chapter in the *Cyclone III Device Handbook*
- *Cyclone II Architecture* chapter in the *Cyclone II Device Handbook*
- *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Stratix II Architecture* chapter in volume 1 in the *Stratix II Device Handbook*
- *Selectable I/O Standards in Stratix II Devices and Stratix II GX Devices* chapter in volume 2 of the *Stratix II Device Handbook*
- *Selectable I/O Standards in Cyclone II Devices* chapter in the *Cyclone II Device Handbook*
- *Selectable I/O Standards in Cyclone II Devices* chapter in the *Cyclone II Device Handbook*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Stratix III Device Handbook*
- *Stratix II Device Handbook*

Document Revision History

Table 9–15 shows the revision history for this chapter.

Table 9–15. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 9–39.	—
May 2007 v7.1.0	• Updated Power Dissipation. • Updated Power-Driven Compilation. • Added Referenced Documents.	Updated figures for the Quartus II software 7.1. Added support for Arria GX.
March 2007 v7.0.0	Minor text edits to include Cyclone III information; added new screenshot to Figure 9-2.	Added support information for the Cyclone III device.
November 2006 v6.1.0	Updated figures to accommodate GUI changes in the software	Added information about Stratix III support. Changes in procedures were made for Quartus II enhancements to new user functionality.
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0: Updated device support Added multi-VCD/SAF support information Updated achieved power reductions values	—
October 2005 v5.1.0	Chapter 9 was formerly Chapter 7 in Volume 1: Stratix II Low Power Design Techniques.	—

10. Analyzing and Optimizing the Design Floorplan

QII52006-7.2.0

Introduction

As FPGA designs grow larger and larger in density, the need to analyze the design for performance, routing congestion, and logic placement to meet the design requirements becomes critical.

You can use the Chip Planner as a powerful tool to perform design analysis and create a design floorplan. With some of the older device families, you must use the Timing Closure Floorplan tool to analyze the device floorplan.

This chapter discusses how to analyze the design floorplan with both the Chip Planner and the Timing Closure Floorplan tools. It also shows you how to create LogicLock™ regions in the design floorplan and how to use the LogicLock design methodology as a performance preservation tool for older device families.

This chapter includes the following topics:

- "Chip Planner Overview" on page 10–2
- "LogicLock Regions" on page 10–6
- "Using LogicLock Regions in the Chip Planner" on page 10–11
- "Design Analysis Using the Chip Planner" on page 10–13
- "Timing Closure Floorplan Overview" on page 10–36
- "Design Analysis Using the Timing Closure Floorplan" on page 10–38
- "Using LogicLock Methodology for Older Device Families" on page 10–57
- "Scripting Support" on page 10–67

To view and create assignments for a design floorplan in supported devices, use the Chip Planner. To make I/O assignments, you must use the Pin Planner tool.



For more information about the Pin Planner tool, refer to the *Early I/O Planning Using the Pin Planner* section of the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Table 10–1 lists the device families supported by the Chip Planner and the Timing Closure Floorplan.

Table 10–1. Chip Planner Device Support

Device Family	Timing Closure Floorplan	Chip Planner
Arria™ GX	—	✓
Stratix® III	—	✓
Stratix II	—	✓
Stratix II GX	—	✓
Stratix	—	✓
Stratix GX	—	✓
Cyclone® III	—	✓
Cyclone II	—	✓
Cyclone	—	✓
HardCopy® II	—	✓
MAX® II	—	✓
MAX 7000	✓	—
ACEX®	✓	—
APEX™ II APEX 20KC APEX 20KE	✓	—
FLEX 10K® FLEX® 10KA FLEX 10KE FLEX 6000	✓	—

This chapter describes how to use the Chip Planner—and the Timing Closure Floorplan for devices not supported by the Chip Planner—for design analysis.

Chip Planner Overview

The Chip Planner provides a visual display of chip resources. It can show logic placement, LogicLock and custom regions, relative resource usage, detailed routing information, fan-in and fan-out paths between registers, and delay estimates for paths. With the Chip Planner, you can view critical path information, physical timing estimates, and routing congestion.

The Chip Planner can perform assignment changes, such as creating and deleting resource assignments, as well as post-compilation changes, such as creating, moving, and deleting logic cells and I/O atoms. By using the

Chip Planner in conjunction with the Resource Property Editor, you can change connections between resources and make post-compilation changes to the properties of logic cells, I/O elements, PLLs, and RAM and DSP blocks. With the Chip Planner, you can view and create assignments for a design floorplan, perform power and design analyses, and implement ECOs in a single tool.

For details about how to implement ECOs in your design using the Chip Planner tool, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Starting Chip Planner

To start the Chip Planner, on the Tools menu, click **Chip Planner (Floorplan & Chip Editor)**. Other methods to start the Chip Planner include:

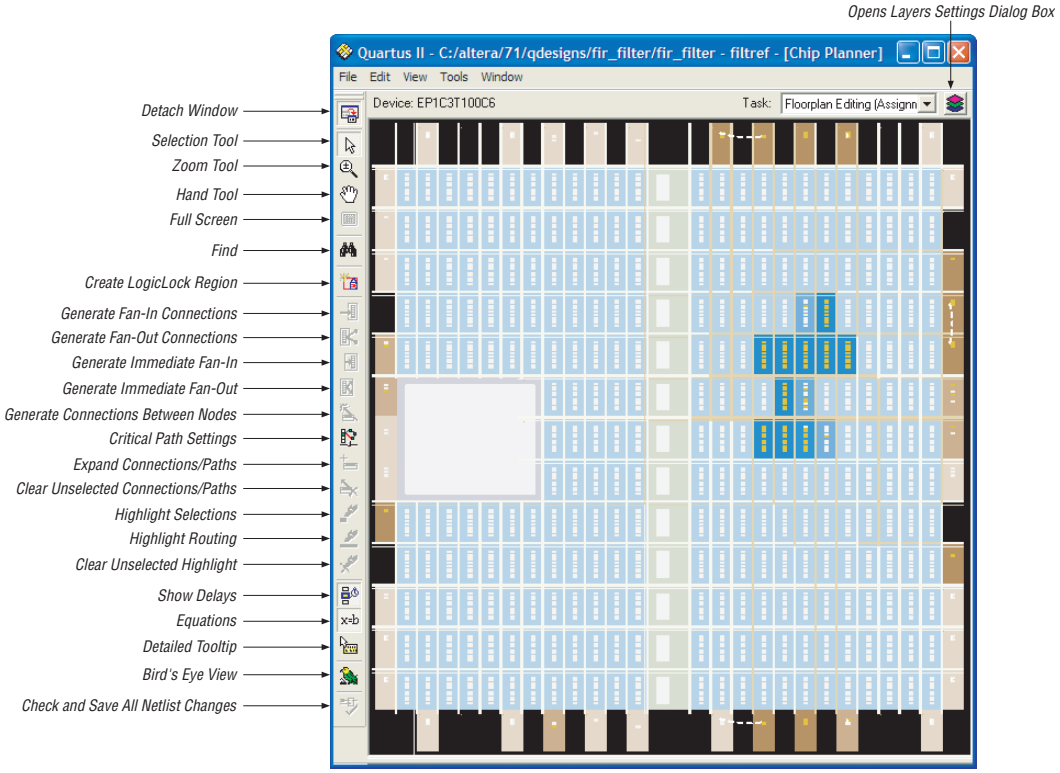
- Click the Chip Planner icon on the Quartus II software toolbar
- Use the shortcut menu from the following sources and use the Locate menu:
  - Compilation Report
  - Project Navigator window
  - RTL source code
  - Node Finder
  - Simulation Report
  - RTL Viewer

If the device in your project is not supported by the Chip Planner and you attempt to start the Chip Planner, the following message appears: `Can't display Chip Planner: the current device family is unsupported.` Use the Timing Closure Floorplan for these devices.

Chip Planner Toolbar

The Chip Planner gives you powerful capabilities for design analysis with a very user-friendly GUI. Many functions within the Chip Planner can be selected or performed from the menu items or by clicking the icons on the toolbar. Figure 10–1 shows an example of the Chip Planner Toolbar and provides descriptions for commonly used icons available from the Chip Planner toolbar.

Figure 10–1. Chip Planner Toolbar





You can customize the icons present on the Chip Planner's toolbar by clicking **Customize Chip Planner** on the Tools menu (if the Chip Planner window is attached) or by clicking **Customize** on the Tools menu (if the Chip Planner window is detached).

Chip Planner Tasks and Layers

The Chip Planner has predefined tasks that enable you to quickly implement ECO changes or manipulate assignments for the floorplan of the device. To select a task, click on it in the Task pull-down menu. The predefined tasks in the Chip Planner are:

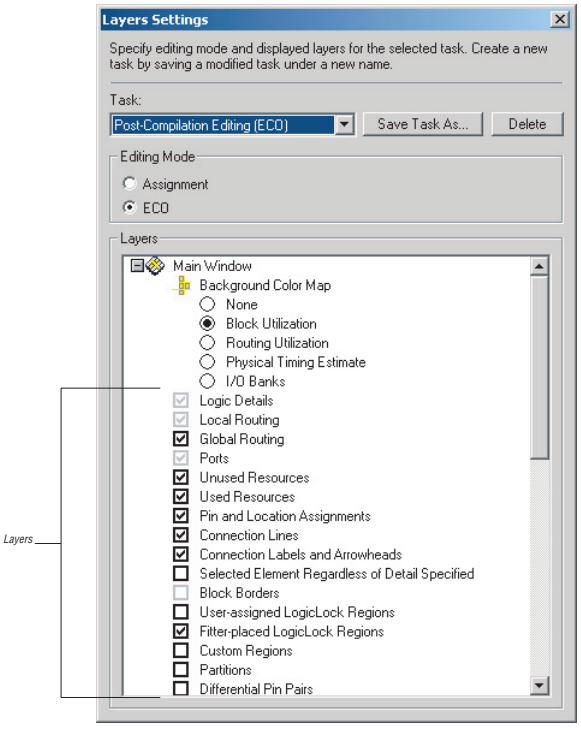
- Floorplan Editing (Assignment)
- Post-Compilation Editing (ECO)
- Partition Display (Assignment)

- Global Clock Network (Assignment)
- Power Analysis (Assignment) — available for Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, and HardCopy II devices only

In the Chip Planner, layers allow you to specify the graphic elements that are displayed for a given task. You can turn off the display of specific graphic elements to increase the window refresh speed and reduce visual clutter when viewing complex designs. The Background Color Map indicates the relative level of resource usage for different areas of the device. For example, Routing Utilization indicates the relative routing utilization, and Physical Timing Estimate indicates the relative physical timing.

Each predefined task in the Chip Planner has a Background Color Map, a set of displayed layers, and an editing mode associated with it. Click the **Layers** icon (shown in Figure 10–1) to display the Layers Settings window (Figure 10–2). In this window you can select the layers and background color map for each task.

Figure 10–2. Layers in the Chip Planner



The Chip Planner operates in either **Assignment** or **ECO** mode. You can use either of these modes to perform design analyses. Use the Floorplan Editor (Assignment) task in the Assignment mode to manipulate LogicLock regions and location assignments in your design. The Post Compilation Changes (ECO) task in ECO mode allows you to implement ECO changes in your design. The Partition Display (Assignment) task allows you to view the placement of nodes and color-codes the nodes based on their partition. When you select the Global Clock Network (Assignment) task, you can see all the global clock regions in your device. The Power Analysis (Assignment) task allows you to view high and low power resources in a Stratix III device.



For more information about the ECO mode of operation, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

You can also create and save your own custom tasks. When you create a custom task, you can turn on or off any layer in your task. Layers can be turned on or off by checking the appropriate box located next to each layer. You can also select different Background Color Maps to be used for your custom task. After selecting the required settings, you can click **Save Task As** to save your custom task.

LogicLock Regions

LogicLock regions are user-defined rectangular regions within the device. You can use LogicLock regions to create a floorplan for your design. Your floorplan may contain several non-overlapping LogicLock regions. A LogicLock region is defined by its size (height and width) and location (where the region is located on the device). You can specify the size or location of a region, or both, or the Quartus® II software can generate these properties automatically. The Quartus II software bases the size and location of a region on the region's contents and the module's timing requirements. Table 10–2 describes the options for creating LogicLock regions.

Table 10–2. Types of LogicLock Regions (Part 1 of 2)

Properties	Values	Behavior
State	Floating (default), Locked	Floating regions allow the Quartus II software to determine the region's location on the device. Locked regions represent user-defined locations for a region and are shown with a solid boundary in the floorplan. A locked region must have a fixed size.
Size	Auto (default), Fixed	Auto-sized regions allow the Quartus II software to determine the appropriate size of a region given its contents. Fixed regions have a user-defined shape and size.

Table 10–2. Types of LogicLock Regions (Part 2 of 2)

Properties	Values	Behavior
Reserved	Off (default), On	The reserved property allows you to define whether the Fitter can use the resources within a region for entities that are not assigned to the region. If the reserved property is turned on, only items assigned to the region can be placed within its boundaries.
Soft	Off (default), On	Soft (on) regions give more deference to timing constraints, and allow some entities to leave a region if it improves the performance of the overall design. Hard (off) regions do not allow contents to be placed outside of the boundaries of the region.
Origin	Any Floorplan Location	The origin is the origin of the LogicLock region's placement on the floorplan. For Arria GX devices, Stratix and Cyclone series devices, and MAX II devices, the origin is located in the lower left corner. For other Altera device families, the origin is located in the upper left corner.

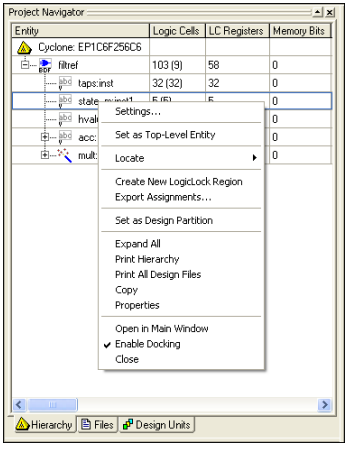


The Quartus II software cannot automatically define a region's size if the location is locked. Therefore, if you want to specify the exact location of the region, you must also specify the size. Mercury devices support only locked and fixed regions.

Creating LogicLock Regions

After you perform either a full compilation or analysis and elaboration on the design, the Quartus II software displays the hierarchy of the design. On the View menu, click **Project Navigator**. With the hierarchy of the design fully expanded, as shown in Figure 10–3, right-click on any design entity in the design and click **Create New LogicLock Region** to create a LogicLock region.

Figure 10–3. Using the Project Navigator to Create LogicLock Regions



Placing LogicLock Regions

A fixed region must contain all resources required for the module. Although the Quartus II software can automatically place and size LogicLock regions to meet resource and timing requirements, you can manually place and size regions to meet your design needs. To do so, follow these guidelines:

- LogicLock regions with pin assignments must be placed on the periphery of the device, adjacent to the pins. (For Stratix and Cyclone series of devices and MAX II devices, you must also include the I/O block).
- Floating LogicLock regions cannot overlap.
- Avoid creating fixed and locked regions that overlap.



If you want to import multiple instances of a module into a top-level design, you must ensure that the device has two or more locations with exactly the same device resources. If the device does not have another area with exactly the same resources, the Quartus II software generates a fitting error during compilation of the top-level design.

Placing Device Features into LogicLock Regions

A LogicLock region includes all device resources within its boundaries, including memory and pins. You can assign pins to LogicLock regions; however, this placement puts location constraints on the region. When the Quartus II software places a floating auto-sized region, it places the region in an area that meets the requirements of the LogicLock region's contents.



Pin assignments to LogicLock regions are effective only in fixed and locked regions. Pins assigned to floating regions do not influence the region's placement.

Only one LogicLock region can claim a device resource. If the boundary includes part of a device resource, the Quartus II software allocates the entire resource to the LogicLock region.

LogicLock Regions Window

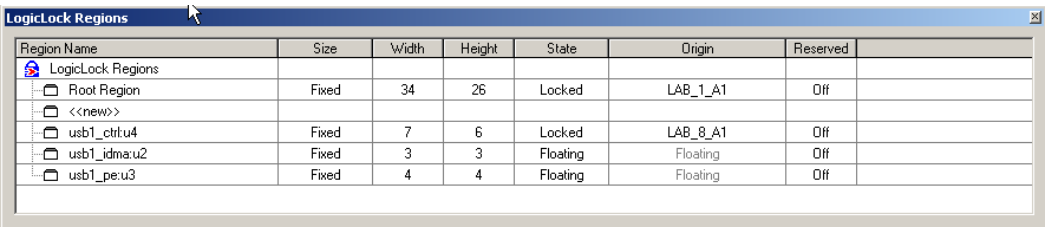
The LogicLock window consists of the LogicLock Regions window (Figure 10–4) and the **LogicLock Region Properties** dialog box. Use the LogicLock Regions window to create LogicLock regions and assign nodes and entities to them. The dialog box provides a summary of all LogicLock regions in your design. In the LogicLock Regions window, you can modify a LogicLock region's size, state, width, height, and origin as well

as whether the region is soft or reserved. When the region is back-annotated, the placement of each node within the region is relative to the region's origin, and the region's node placements are maintained during subsequent compilations.



The origin location varies based on device family. For Arria GX devices, Stratix and Cyclone series devices, and MAX II devices, the LogicLock region's origin is located at the lower left corner of the region. For all other supported devices, the origin is located at the upper left corner of the region.

Figure 10–4. LogicLock Regions Window



You can customize the LogicLock Regions window by dragging and dropping the various columns. Columns can also be hidden.



The **Soft** and **Reserved** columns are not shown by default.

For designs targeting Stratix and Cyclone series and MAX II devices, the Quartus II software automatically creates a LogicLock region that encompasses the entire device. This default region is labelled `Root_region`, and it is effectively locked and fixed.

Use the **LogicLock Region Properties** dialog box to obtain detailed information about your LogicLock region, such as which entities and nodes are assigned to your region and what resources are required. The **LogicLock Region Properties** dialog box shows the properties of the current selected regions.



To open the **LogicLock Region Properties** dialog box, double-click any region in the LogicLock Regions window, or right-click the region and click **Properties**.

Excluded Resources

The Excluded Resources feature allows you to easily exclude specific device resources such as DSP blocks or M4K memory blocks from a LogicLock region. For example, you can specify resources that belong to a specific entity that are assigned to a LogicLock region, and specify that these resources be included with the exception of the DSP blocks. Use the Excluded Resources feature on a per-LogicLock region member basis.

To exclude certain device resources from an entity, in the **LogicLock Region Properties** dialog box, highlight the entity in the **Design Element** column, and click **Edit**. In the **Edit Node** dialog box, under **Excluded Element Types**, click on **(...)**. In the **Excluded Resources Element Types** dialog box, you can select the device resources you want to exclude from the entity. Once you have selected the resources to exclude, the **Excluded Resources** column is updated in the **LogicLock Region Properties** dialog box to reflect the excluded resources.



The Excluded Resources feature prevents certain resource types from being included in a region, but it does not prevent the resources from being placed inside the region unless the region's "Reserved" property is set to **On**. To inform the Fitter that certain resources are not required inside a LogicLock region, define a resource filter.

Hierarchical (Parent and Child) LogicLock Regions

You can define a hierarchy for a group of regions by declaring parent and child regions. The Quartus II software places a child region completely within the boundaries of its parent region, allowing you to further constrain module locations. Additionally, parent and child regions allow you to further improve a module's performance by constraining the nodes in the module's critical path.

To make one LogicLock region a child of another LogicLock region, in the LogicLock Regions window, select the new child region and drag and drop it in its new parent region.



The LogicLock region hierarchy does not have to be the same as the design hierarchy.

A child region's location can float within its parent or remain locked relative to its parent's origin. A locked parent region's location is locked relative to the device. If the child's location is locked and the parent's location is changed, the child's origin changes but maintains the same

placement relative to the origin of its parent. Either you or the Quartus II software can determine a child region's size; however, the child region must fit entirely within the parent region.

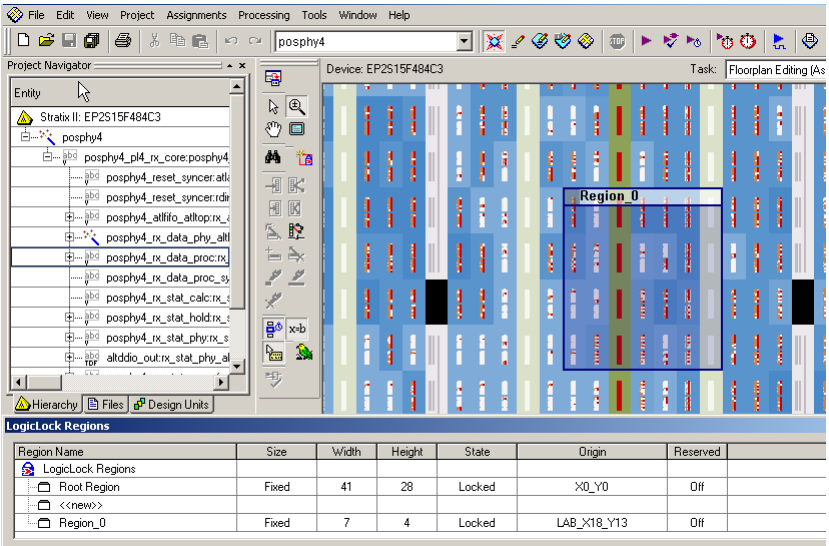
Using LogicLock Regions in the Chip Planner

Assigning LogicLock Region Content

Once you have defined a LogicLock region, you must assign resources to it using the Chip Planner, the **LogicLock Regions** dialog box, or a Tcl script.

You can drag selected logic displayed in the **Hierarchy** tab of the **Project Navigator**, in the Node Finder, or in a schematic design file, and drop it into the Chip Planner or the **LogicLock Regions** dialog box. Figure 10–5 shows logic that has been dragged from the **Hierarchy** tab of the **Project Navigator** and dropped into a LogicLock region in the **Chip Planner**.

Figure 10–5. Drag and Drop Logic in the Chip Planner



You can also drag logic from the **Hierarchy** tab of the **Project Navigator** and drop it in the **LogicLock Regions Properties** dialog box. Logic can also be dropped into the **Design Element Assigned** column of the **Contents** tab of the **LogicLock Region Properties** box.



You must assign pins to a LogicLock region manually. The Quartus II software does not include pins automatically when you assign an entity to a region. The software only obeys pin assignments to locked regions that border the periphery of the device. For the Stratix and Cyclone series of devices and MAX II devices, the locked regions must include the I/O pins as resources.

Creating LogicLock Regions with the Chip Planner

This section explains the basics of creating LogicLock regions. You can use the following methods s to create a LogicLock region in the Chip Planner:

- On the Assignments menu, click **LogicLock Regions Window**.
- On the View menu, click **Project Navigator**. Use the **Hierarchy** tab.
- In the View menu of the Chip Planner, click the **Create LogicLock Region** tab.
- Use a Tcl script.

Viewing Connections Between LogicLock Regions in the Chip Planner

You can view and edit LogicLock regions using Chip Planner. Select the task **Floorplan Editing (Assignment)** or select any custom task with "Assignment" as the editing mode to manipulate LogicLock regions.

The Chip Planner shows the connections between LogicLock regions. Rather than showing multiple connection lines from one LogicLock region to another, you can select the option of viewing connections between LogicLock regions as a single bundled connection. To use this option, open the Chip Planner floorplan, and on the View menu, click **Generate Inter-region Bundles**.

In the **Generate Inter-region Bundles** dialog box, specify the **Source node to region fanout less than** and the **Bundle width greater than** values.



For more information about the parameters in the **Generate Inter-region Bundles** dialog box, refer to the Quartus II Help.

Design Analysis Using the Chip Planner

The Chip Planner assists you in visually analyzing your design at any stage of your design cycle. With the Chip Planner tool, you can view post-compilation placement, connections, and routing paths. You can also create LogicLock regions and location assignments. The Chip Planner allows you to create new logic cells and I/O atoms and to move existing logic cells and I/O atoms using the architectural floorplan of your design. You can also see global and regional clock regions within the device, and the connections between both I/O atoms and PLLs and the different clock regions.

From the Chip Planner, you can launch the Resource Property Editor. The Resource Property Editor lets you make changes to the properties and parameters of device resources, and modify connectivity between certain types of device resources. Changes that you perform are logged within the Change Manager. The Change Manager helps you track the various changes you make on your design floorplan, so that you can selectively undo the changes if necessary.



For more information about the Resource Property Editor and the Change Manager, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

The following sections present design analysis procedures and views of Chip Planner, which can be used with any predefined task of the Chip Planner (unless explicitly stated that a certain procedure requires a particular task or editing mode).

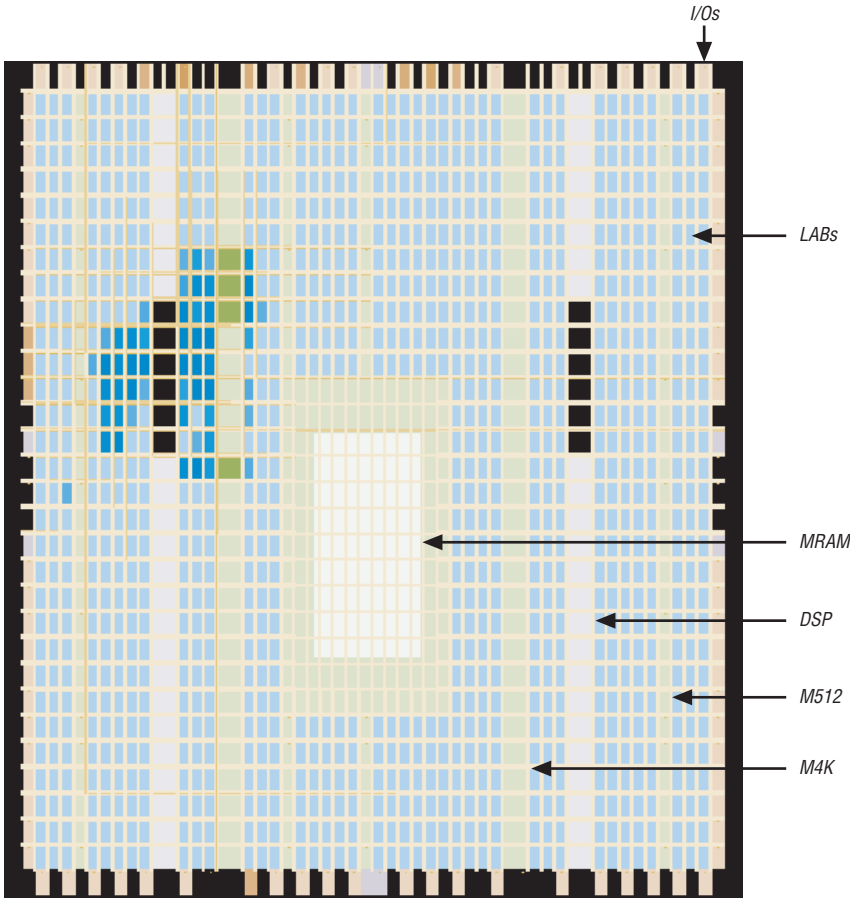
Chip Planner Floorplan Views

The Chip Planner uses a hierarchical zoom viewer that shows various abstraction levels of the targeted Altera device. As you increase the zoom level, the level of abstraction decreases, thus revealing more detail about your design.

First-Level View

The first level provides a high-level view of the entire device floorplan. You can locate and view the placement of any node in your design. Figure 10–6 shows the Chip Planner's Floorplan first level view of a Stratix device.

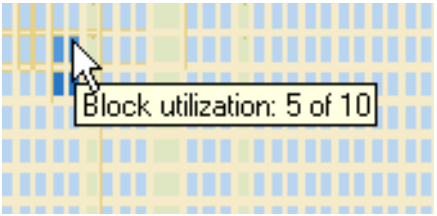
Figure 10–6. Chip Planner's First-Level Floorplan View



Each resource is shown in a different color. The Chip Planner floorplan uses a gradient color scheme in which the color becomes darker as the utilization of a resource increases. For example, as more LEs are used in the LAB, the color of the LAB becomes darker.

When you place the mouse pointer over a resource at this level, a tooltip appears that describes, at a high level, the utilization of the resource (Figure 10–7).

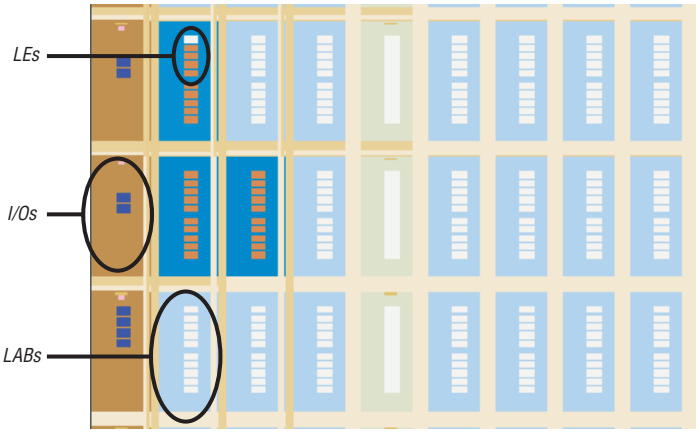
Figure 10–7. Tooltip Message: First-Level View



Second-Level View

As you zoom in, the level of detail increases. Figure 10–8 shows the second-level view of the Chip Planner Floorplan for a Stratix device.

Figure 10–8. Chip Planner's Second Level Floorplan View



At this level, the contents of LABs and I/O banks and the routing channels that are used to connect resources are all visible.

When you place the mouse pointer over an LE or ALM at this level, a tooltip is displayed (Figure 10–9) that shows the name of the LE/ALM, the location of the LE/ALM, and the number of resources that are used with that LAB. When you place the mouse pointer over an interconnect, the tooltip shows the routing channels that are used by that interconnect. At this level, you can move LEs, ALMs, and I/Os from one physical location to another.

Figure 10–9. Tooltip Message: Second-Level View



Third-Level View

At the third level, which provides a more detailed view, you can see each routing resource that is used within a LAB in the FPGA. Figure 10–10 shows the level of detail at the third level view for a Stratix device.

From the third level, you can move LEs, ALMs, and I/Os from one physical location to another. You can move a resource by selecting, dragging, and dropping it into the desired location. At this level, you can also create new LEs and I/Os when you are in the post-compilation (ECO) mode.



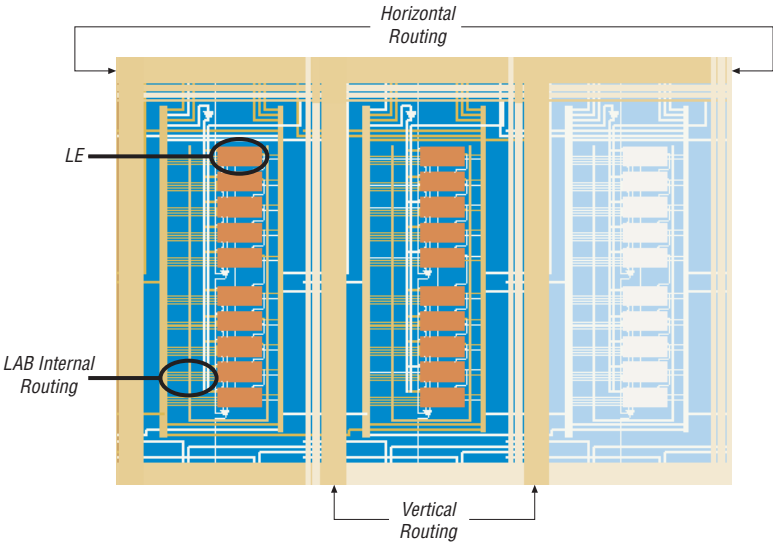
You can delete a resource only after all of its fan-out connections are removed. Moving nodes in the Floorplan Editing (Assignment) task creates an assignment. However, if you move logic nodes in the Post-Compilation Editing (ECO) task, that change is considered an ECO change.

For more information about Floorplan Assignments, refer to "Viewing Assignments in the Chip Planner" on page 10–32.



For more information about performing ECOs, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

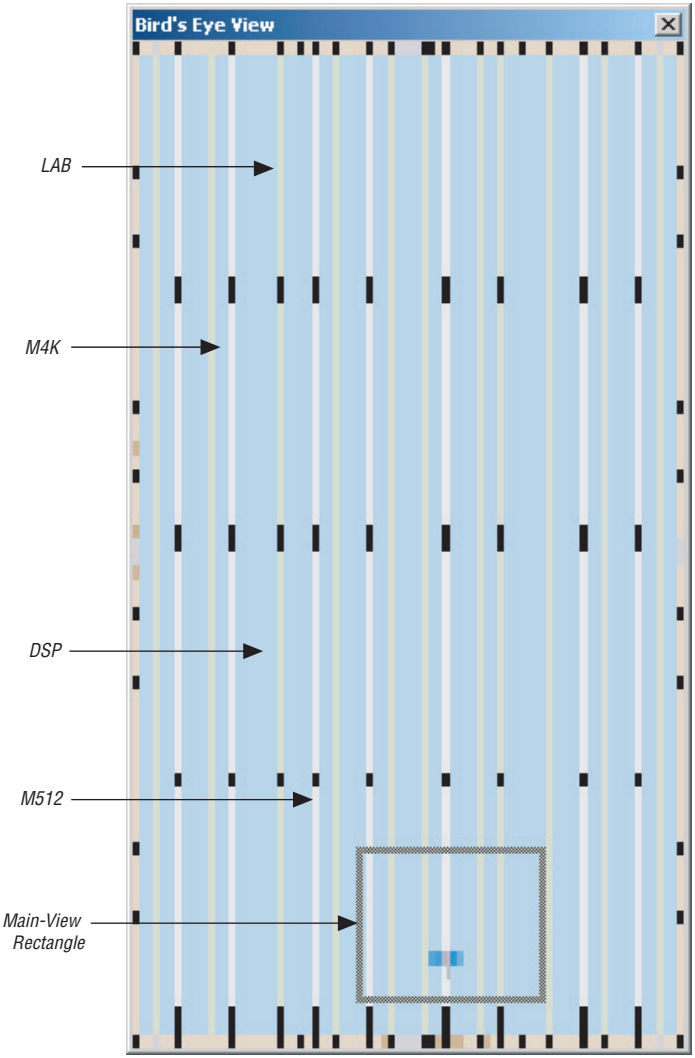
Figure 10–10. Chip Planner's Third Level Floorplan View



Bird's Eye View

The Bird's Eye View (Figure 10–11) displays a high-level picture of resource usage for the entire chip and provides a fast and efficient way to navigate between areas of interest in the Chip Planner.

Figure 10–11. Bird's Eye View



The Bird's Eye View is displayed as a separate window that is linked to the Chip Planner floorplan. When you select an area of interest in the Bird's Eye View, the Chip Planner floorplan automatically refreshes to show that region of the device. As you change the size of the main-view rectangle in the Bird's Eye View window, the main Chip Planner

floorplan window also zooms in (or zooms out). You can make the main-view rectangle smaller in the Bird's Eye View to see more detail on the Chip Planner floorplan window.

The Bird's Eye View is particularly useful when parts of your design that you are interested in are at opposite ends of the chip, and you want to quickly navigate between resource elements without losing your frame of reference.

Viewing Architecture-Specific Design Information

With the Chip Planner, you can view the following architecture-specific information related to your design:

- **Device routing resources used by your design**—View how blocks are connected, as well as the signal routing that connects the blocks.
- **LE configuration**—View how a logic element (LE) is configured within your design. For example, you can view which LE inputs are used; if the LE utilizes the register, the look-up table (LUT), or both; as well as the signal flow through the LE.
- **ALM configuration**—View how an adaptive logic module (ALM) is configured within your design. For example, you can view which ALM inputs are used, if the ALM utilizes the registers, the upper LUT, the lower LUT, or all of them. You can also view the signal flow through the ALM.
- **I/O configuration**—View how the device I/O resources are used. For example, you can view which components of the I/O resources are used, if the delay chain settings are enabled, which I/O standards are set, and the signal flow through the I/O.
- **PLL configuration**—View how a phase-locked loop (PLL) is configured within your design. For example, you can view which control signals of the PLL are used with the settings for your PLL.
- **Timing**—View the delay between the inputs and outputs of FPGA elements. For example, you can analyze the timing of `DATAB` input to the `COMBOUT` output.

In addition, you can modify the following properties of an Altera device with the Chip Planner:

- LEs and ALMs
- I/O cells
- PLLs
- Registers in RAM and DSP blocks
- Connections between elements
- Placement of elements

For more information about LEs, ALMs, and other resources of an FPGA device, refer to the relevant device handbook.

Viewing Critical Paths

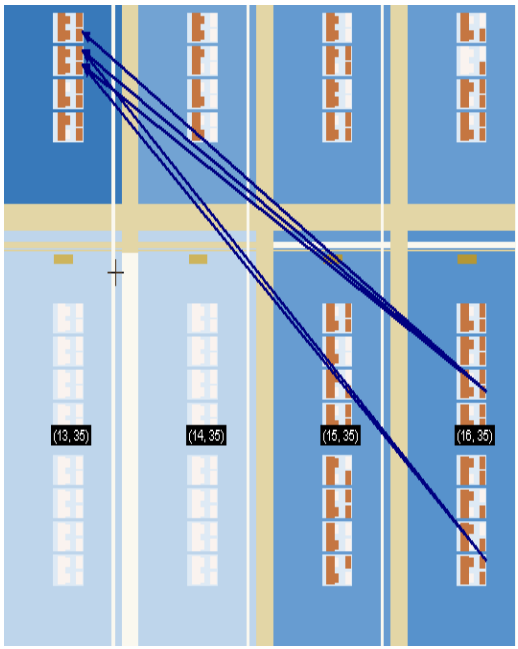
Critical paths are timing paths in your design that have a negative slack. These timing paths can span from device I/Os to internal registers, registers-to-registers, or registers-to-devices I/Os. The View Critical Paths feature displays routing paths in the Chip Planner, as shown in Figure 10–12. The criticality of a path is determined by its slack and is shown in the timing analysis report. Design analysis for timing closure is a fundamental requirement for optimal performance in highly complex designs. The Quartus II Chip Planner helps you close timing on complex designs with its analysis capability.

Viewing critical paths in the Chip Planner helps you analyze why a specific path is failing. You will be able to see if any modification in the placement can potentially reduce the negative slack.

To view critical paths in the Chip Planner while using the Quartus II Classic Timing Analyzer, on the View menu, click **Critical Path Settings**. In the **Critical Path Settings** dialog box, click **Show Path** (see Figure 10–13 on page 10–22).

If you are using the TimeQuest Timing Analyzer to locate the critical paths, run the Report Timing task from the Custom Reports group in the Tasks pane of the TimeQuest GUI. From the View pane, which lists the failing paths, you can right-click on any failing path or node, and select **Locate Path**. From the pop-up dialog box, select **Chip Planner** to see the failing path in the Chip Planner.

Figure 10–12. Chip Planner Showing Critical Path



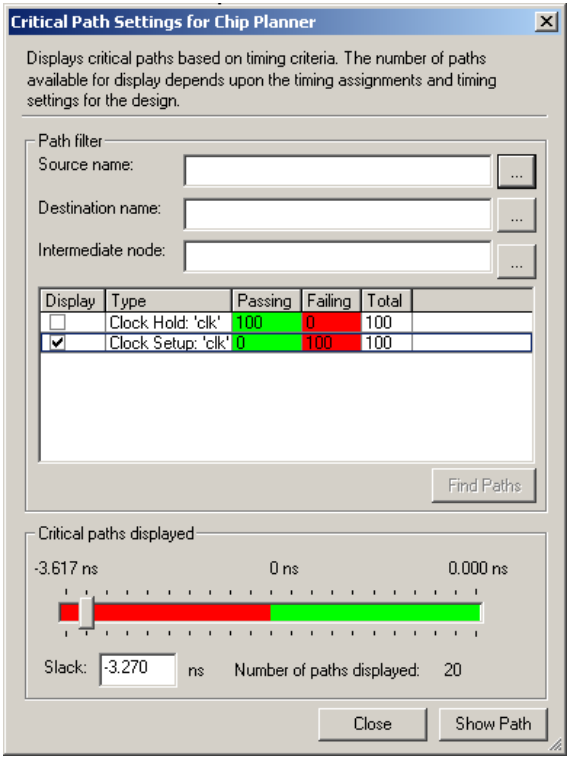
When viewing critical paths, you can specify the clock in the design you want to view. You determine the paths to be displayed by specifying the slack threshold in the slack field of the **Critical Path Settings** dialog box. This dialog box also helps you to filter specific paths based on the source and destination registers.

Timing settings must be made and a timing analysis performed for paths to be displayed in the floorplan.

For more information about performing static timing analysis with the Quartus II Classic Timing Analyzer, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

For more information about performing static timing analysis with the Quartus II TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Figure 10–13. Critical Path Settings for the Chip Planner



Viewing Physical Timing Estimates

In the Chip Planner, you can select a resource and see the approximate delay to any other resource on the device. After you select a resource, the delay is represented by the color of potential destination resources. The lighter the color of the resource, the longer the delay.

To see the physical timing map of the device, in the Chip Planner, click the **Layers** icon located next to the **Task** pull-down menu. Under the Background Color Map, select **Physical Timing Estimate**. Select a source and move your cursor to a destination resource. The Chip Planner displays the approximate routing delay between your selected source and destination register.

You can use the physical timing estimate information when attempting to improve the Fitter results by manually moving logic in a device or when creating LogicLock regions to group logic together. This feature allows

you to estimate the physical routing delay between different nodes so that you can place critical nodes and modules closer together, and move non-critical or unrelated nodes and modules further apart.

In addition to reducing delay between critical nodes, you can make placement assignments to reduce the routing congestion between critical and noncritical entities and modules. This allows the Quartus II Fitter to meet the design timing requirements.



Note that moving logic and creating manual placements is an advanced technique to meet timing requirements and must be done after careful analysis of the design. Moving nodes in the Floorplan Editing (Assignment) task creates an assignment. However, if you move logic nodes in the Post-Compilation Editing (ECO) task, that change is considered an ECO change.

For more information about Floorplan Assignments, refer to "Viewing Assignments in the Chip Planner" on page 10–32.



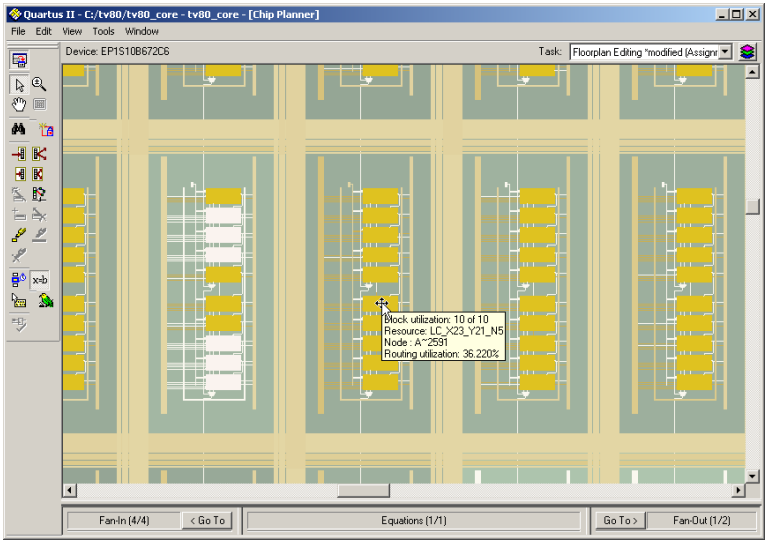
For more information about performing ECOs, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Viewing Routing Congestion

The Routing Congestion view allows you to determine the percentage of routing resources used after a compilation. This feature identifies where there is a lack of routing resources. This information helps you to make decisions about design changes that may be necessary to ease the routing congestion and thus meet design requirements. The congestion is visually represented by the color and shading of logic resources. The darker shading represents a greater routing resource utilization. To view the routing congestion in the Chip Planner, click the **Layers** icon located next to the **Task** menu. Under **Background Color Map,** select the **Routing Utilization** map (Figure 10–14).

If you intend to use a HardCopy II device, there is no selection for Background Color. To get to the routing congestion view with HardCopy II, turn on **Routing Congestion**.

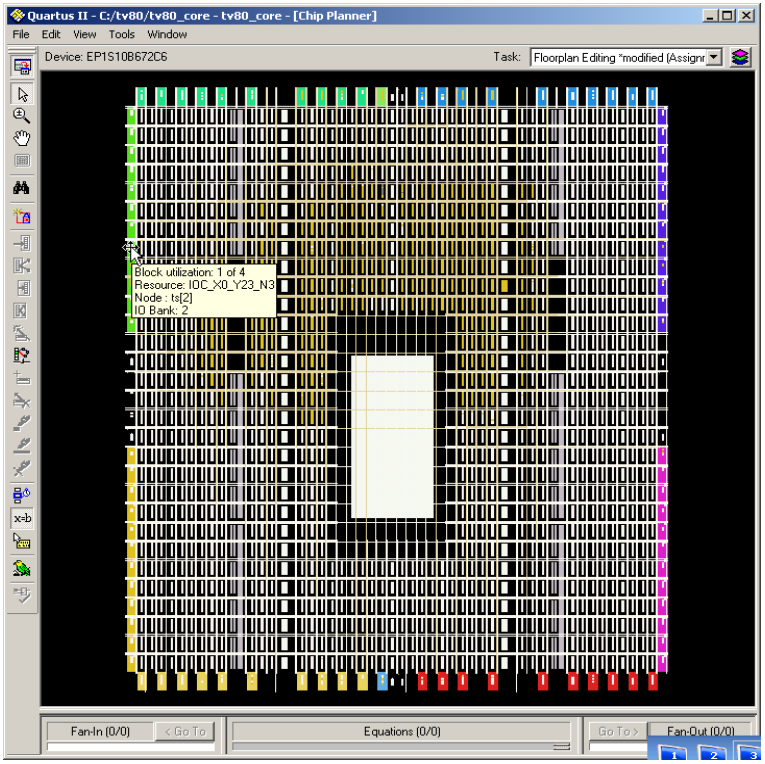
Figure 10–14. Viewing Routing Congestion Map in the Chip Planner



Viewing I/O Banks

The Chip Planner can show all of the I/O banks of the device displayed in different colors. To see the I/O bank map of the device, click the **Layers** icon located next to the **Task** menu. Under **Background Color Map,** select the **I/O Banks** map. See Figure 10–15.

Figure 10–15. Viewing I/O Banks in the Chip Planner

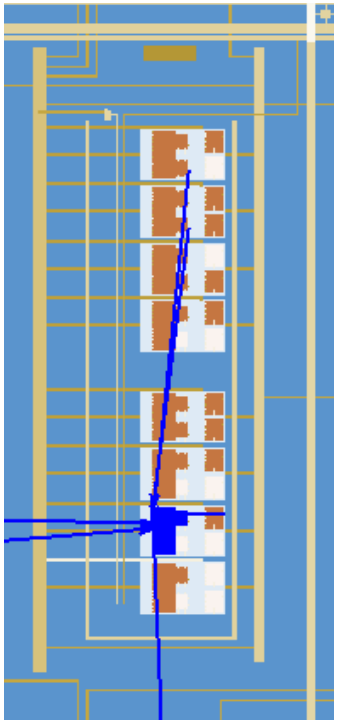


All predefined tasks in the Chip Planner show the Block Utilization Map as the default Background Color Map.

Generating Fan-In and Fan-Out Connections

This feature enables you to view the atoms that fan-in to or fan-out from the selected atom. To remove the connections displayed, use the **Clear Connections** icon in the Chip Planner toolbar. Figure 10–16 shows the fan-in connections for the selected resource.

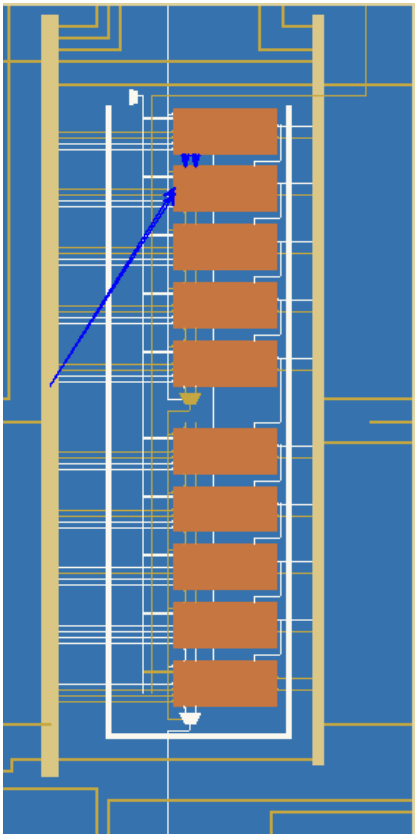
Figure 10–16. Generated Fan-In



Generating Immediate Fan-In and Fan-Out Connections

This feature enables you to view the immediate resource that is the fan-in or fan-out connection for the selected atom. For example, selecting a logic resource and choosing to view the immediate fan-in enables you to see the routing resource that drives the logic resource. You can generate immediate fan-in and fan-outs for all logic resources and routing resources. To remove the connections that are displayed, use the **Clear Connections** icon in the toolbar. Figure 10–17 shows the immediate fan-out connections for the selected resource.

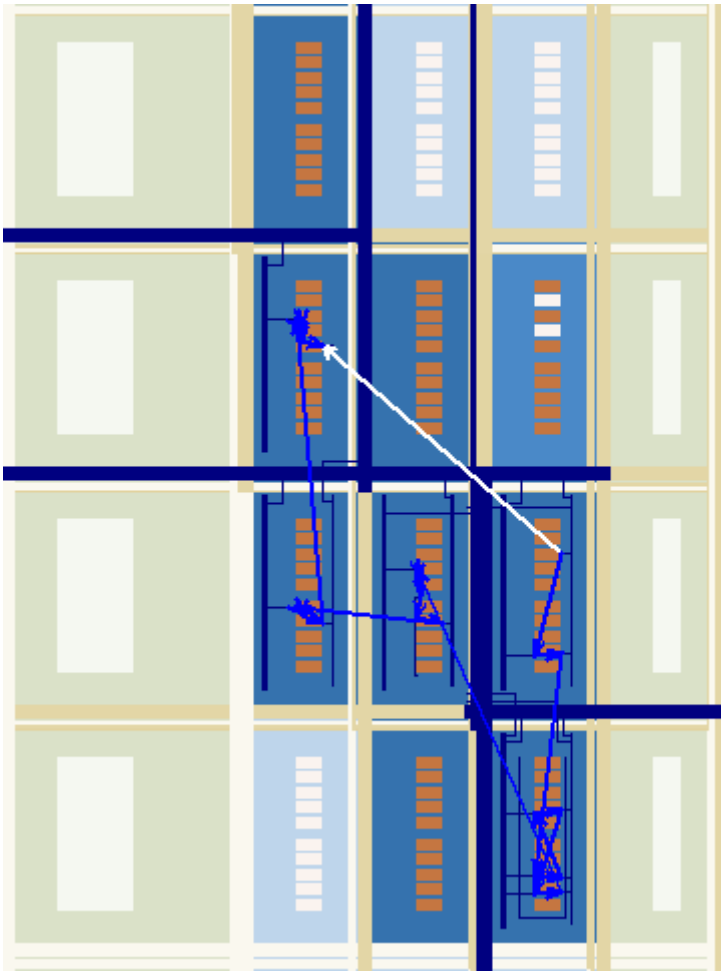
Figure 10–17. Immediate Fan-Out Connection



Highlight Routing

This feature enables you to highlight the routing resources used for a selected path or connection. Figure 10–18 shows the routing resources used between two logic elements.

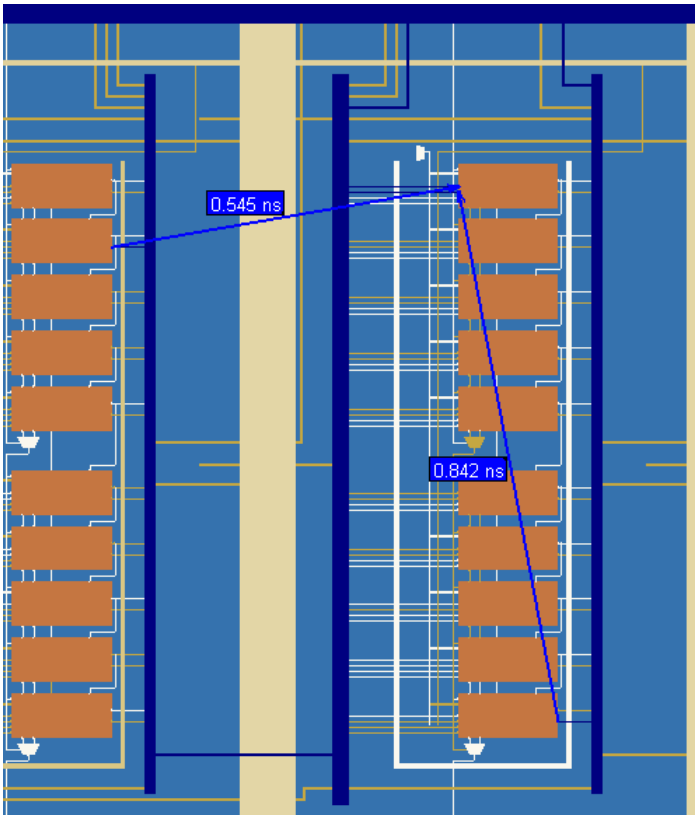
Figure 10–18. Highlight Routing



Show Delays

You can view the timing delays for the highlighted connections when generating connections between elements. For example, you can view the delay between two logic resources or between a logic resource and a routing resource. Figure 10–19 shows the delays between several logic elements.

Figure 10–19. Show Delays



Exploring Paths in the Chip Planner

You can use the Chip Planner to explore paths between logic elements. The following example uses the Chip Planner to traverse paths from the Timing Analysis report.

Locate Path from the Timing Analysis Report to the Chip Planner

To locate a path from the Timing Analysis Report to the Chip Planner, perform the following steps:

1. Select the path you want to locate.
2. Right-click the path in the Timing Analysis Report, right-click **Locate**, then click **Locate in Chip Planner (Floorplan & Chip Editor)** (Figure 10–20).

Figure 10–20. Locate a Timing Path from the Chip Planner Timing Analysis Report

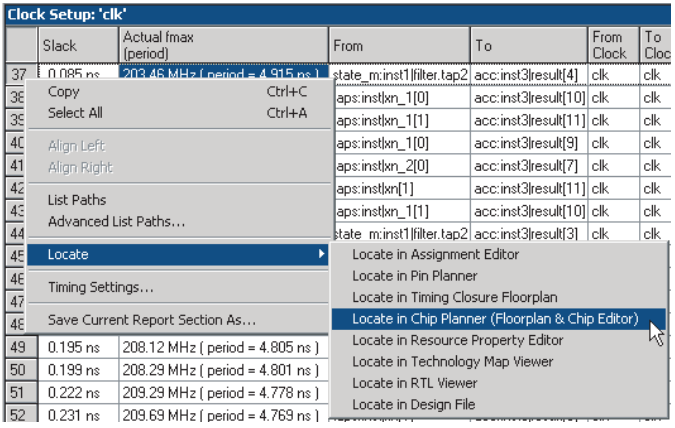
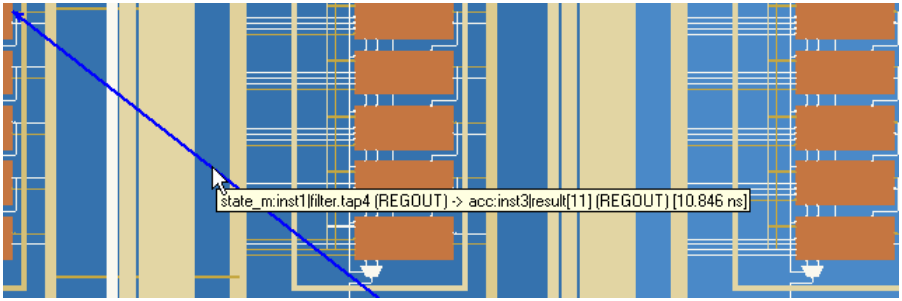


Figure 10–21 shows the path that is displayed in the Chip Planner.

Figure 10–21. Resulting Path

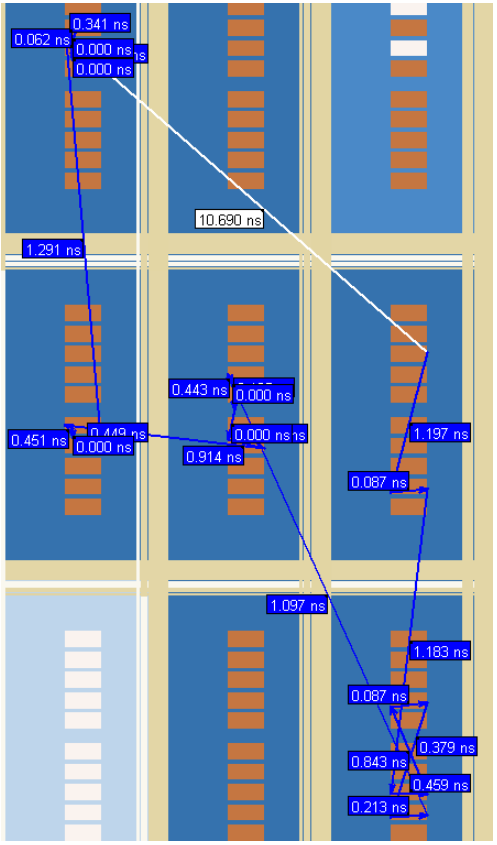


To view the routing resources taken for a path you have located in the Chip Planner, click the **Highlight Routing** icon in the Chip Planner toolbar, or from the View menu, click **Highlight Routing**.

Analyzing Connections for a Path

To determine the connections between items in the Chip Planner, use the **Expand Connections/Paths** icon on the toolbar. To add the timing delays between each connection, use the **Show Delays** icon on the toolbar. Figure 10–22 shows the connections for the selected path that are displayed in the Chip Planner.

Figure 10–22. Path Analysis



Viewing Assignments in the Chip Planner

Location Assignments can be viewed by selecting the appropriate layer set from the tool. To view Location Assignments in Chip Planner, select the task **Floorplan Editing (Assignment)** or any custom task with Assignment editing mode. See Figure 10–23.

The Chip Planner shows location assignments graphically, by displaying assigned resources in a particular color (gray, by default). You can create or move an assignment by dragging the selected resource to a new location.

Figure 10–23. Viewing Assignments in Chip Planner



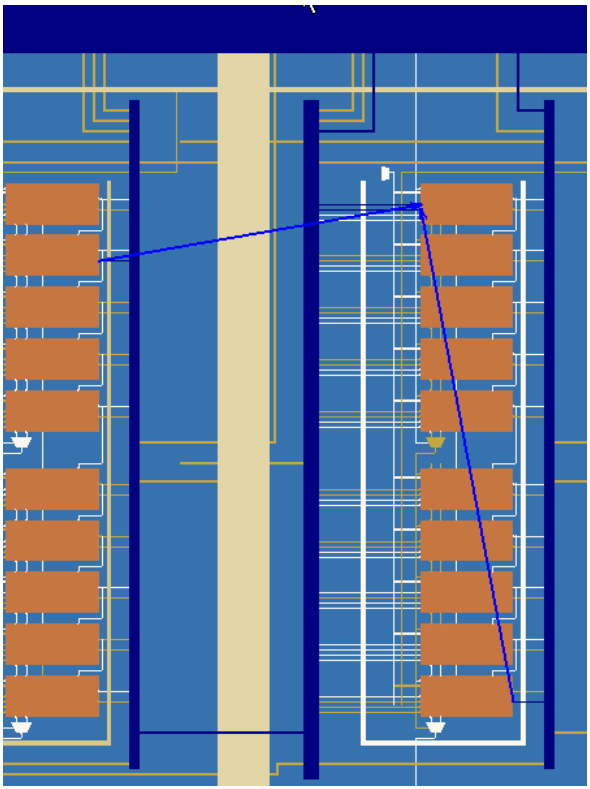
You can make node and pin location assignments and assignments to LogicLock regions and custom regions using the drag-and-drop method in the Chip Planner. The assignments that you create are applied by the Fitter during the next place and route operation.

To learn more about working with location assignments, refer to the Quartus II Help.

Viewing Routing Channels for a Path in the Chip Planner

To determine the routing channels between connections, click the **Highlight Routing** icon on the toolbar. Figure 10–24 shows the routing channels used for the selected path in the Chip Planner.

Figure 10–24. Highlight Routing

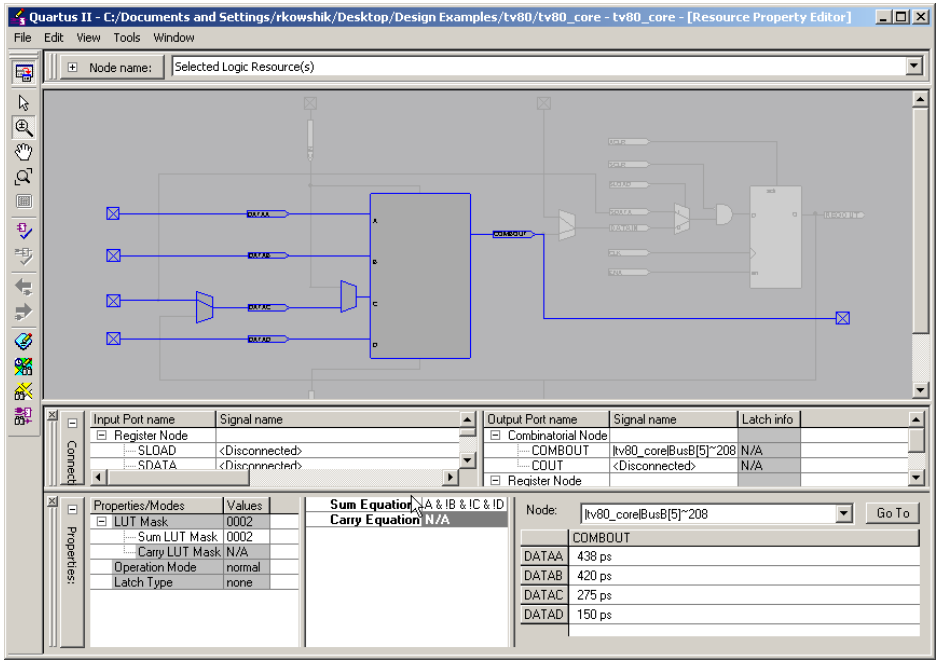


You can view and edit resources in the FPGA using the Resource Property Editor mode of the Chip Planner tool. For more information, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Cell Delay Table

You can view the propagation delay from all inputs to all outputs for any LE in your design. To see the Cell Delay Table for an atom, select the atom in the Chip Planner and right-click. From the pop-up menu, click **Locate** and then click **Locate in Resource Property Editor**. The Resource Property window shows you the atom properties along with the Cell Delay Table, indicating the propagation delay from all inputs to all outputs. Figure 10–25 shows the Cell Delay Table.

Figure 10–25. Cell Delay Table



Viewing High and Low Power Tiles in Stratix III Devices in the Chip Planner

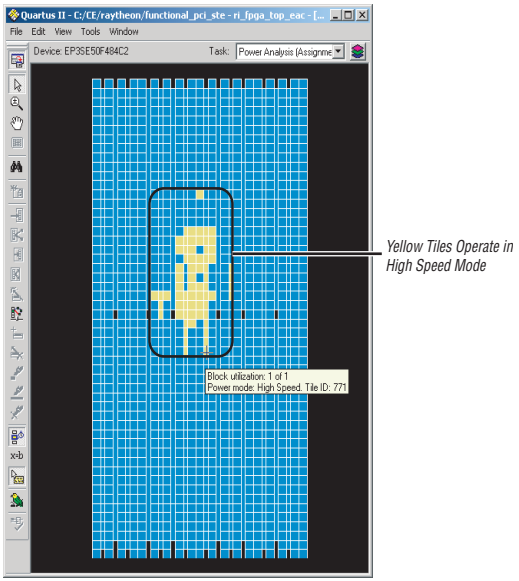
The Chip Planner has a predefined task, **Power Analysis (Assignment)**, which shows the power map of a Stratix III device. Stratix III devices have Adaptive Logic Modules (ALMs) that can operate in either high power mode or low power mode. The power mode is set during the fitting process in the Quartus II software. These ALMs are grouped together to form a larger block, called a "tile".

To learn more about power analyses and optimizations in Stratix III devices, refer to the application note *AN 437: Power Optimization in Stratix III FPGAs*.

When the **Power Analysis (Assignment)** task is selected in the Chip Planner for Stratix III devices, low-power and high-speed tiles are displayed in different colors; yellow tiles operate in a high-speed mode, while blue tiles operate in a low-power mode (see Figure 10–26). The

editing mode in the Power Analysis task is "Assignment." In this mode, you can perform all floor plan-related functions for this task; however, you cannot edit any tiles to change the power mode.

Figure 10–26. Viewing High and Low Power Tiles in a Stratix III Device



Timing Closure Floorplan Overview

For the older device families not supported by the Chip Planner, you can perform the floorplan analysis using the Timing Closure Floorplan. The APEX, ACEX, FLEX, and MAX 7000 families of devices are supported only by the Timing Closure Floorplan. This section explains how to use the Timing Closure Floorplan to enhance your FPGA design analysis.

Table 10–1 on page 10–2 lists the device families supported by the Timing Closure Floorplan Editor and the Chip Planner.

To start the Timing Closure Floorplan Editor, on the Assignments menu, click **Timing Closure Floorplan**.

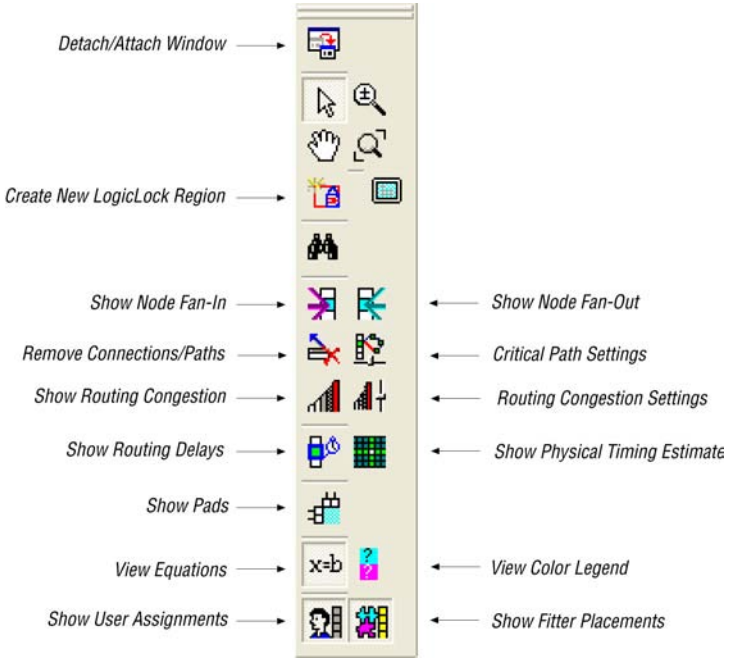
If the device in your project is not supported by the Timing Closure Floorplan, the following message appears: `Can't display a floorplan: the current device family is only supported by Chip Planner.`

If your target device is supported by the Timing Closure Floorplan, you can also start the Timing Closure Floorplan tool by right-clicking any of the following sources, pointing to **Locate**, and clicking **Locate in Timing Closure Floorplan**:

- Compilation Report
- Node Finder
- Project Navigator
- RTL source code
- RTL Viewer
- Simulation Report
- Timing Report

Figure 10–27 shows the icons in the Timing Closure Floorplan toolbar.

Figure 10–27. Timing Closure Floorplan Icons



Design Analysis Using the Timing Closure Floorplan

The Timing Closure Floorplan Editor allows you to analyze your designs visually before and after performing a full design compilation in the Quartus II software. This floorplan editor, used in conjunction with the Quartus II timing analyzer, provides a method for performing design analysis.

Timing Closure Floorplan Views

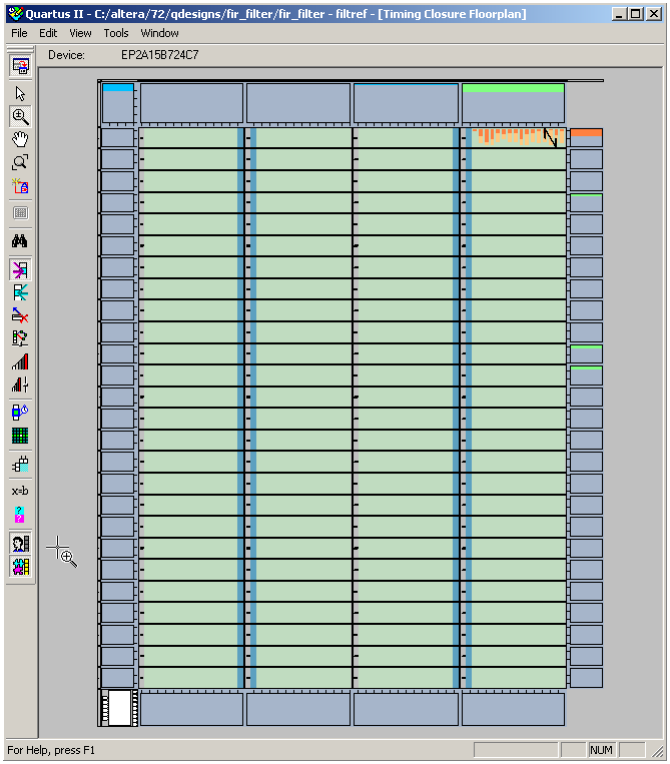
The Timing Closure Floorplan Editor provides the following five views of your design:

- Field view
- Interior Cells view
- Interior Labs view
- Package Top view
- Package Bottom view

Field View

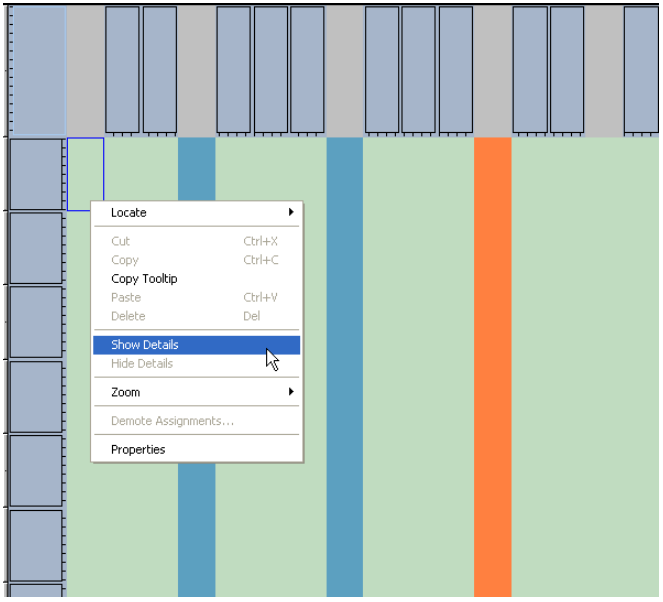
The Field view provides a color-coded, high-level view of the resources used in the device floorplan. All device resources, such as embedded system blocks (ESBs) and MegaLAB blocks, are outlined. Figure 10–28 shows the Field view of an APEX II device.

Figure 10–28. Field View of an APEX II Device



To view the details of a resource in the Field view, select the resource, right-click, and click **Show Details**. To hide the details, select all the resources, right-click, and click **Hide Details** (Figure 10–29).

Figure 10–29. Show and Hide Details of a Logic Array Block in Field View



Other Views

You can also view your design in the Timing Closure Floorplan Editor with the Interior Cells, Interior Labs, Package Top, and Package Bottom views. Use the View menu to display the various floorplan views. The Interior Cells view provides a detailed view of device resources, including device pins and individual logic elements within a MegaLAB.

Viewing Assignments

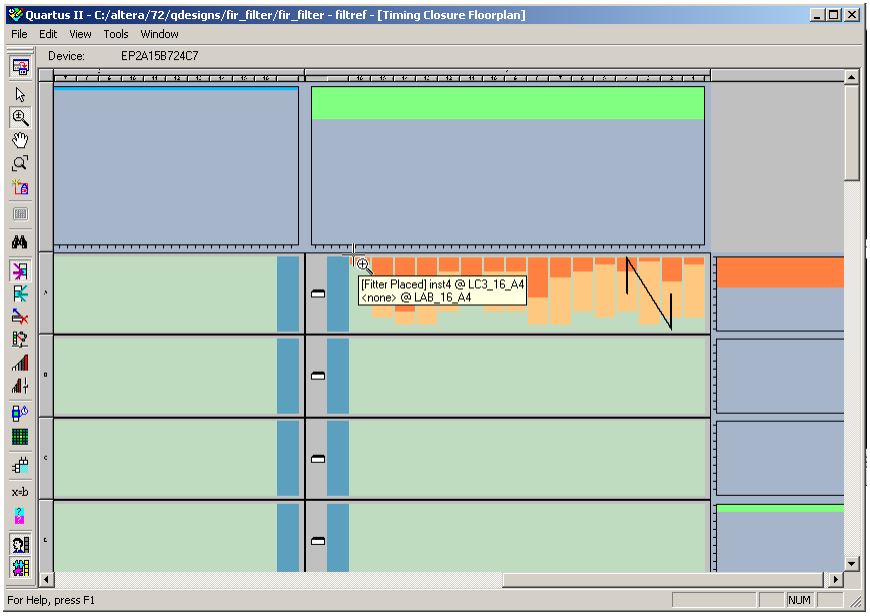
The Timing Closure Floorplan Editor differentiates between user assignments and Fitter placements. User assignments include LogicLock regions and are made by a user.

If the device is changed after a compilation, the user assignment and Fitter placement options cannot be used together. When this situation occurs, the Fitter placement displays the last compilation result and the user assignment displays the floorplan of the newly selected device.

To see the user assignments, click the **Show User Assignments** icon in the Floorplan Editor toolbar, or, on the View menu, point to Assignments and click **Show User Assignments**. To see the Fitter placements, click the

Show Fitter Placements icon in the Floorplan Editor toolbar, or, on the View menu, point to Assignments and click **Show Fitter Placements**. Figure 10–30 shows the Fitter placements.

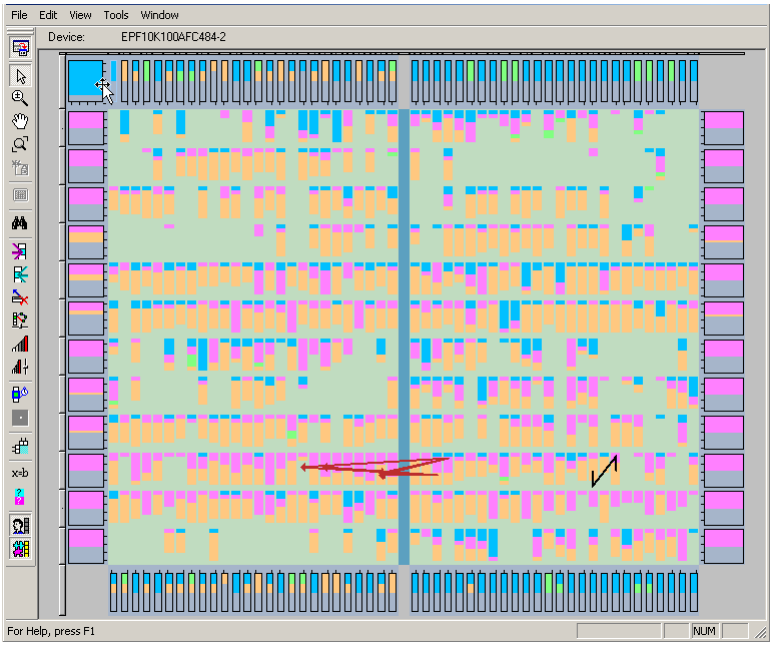
Figure 10–30. Fitter Placements



Viewing Critical Paths

The View Critical Paths feature displays routing paths in the floorplan, as shown in Figure 10–31. The criticality of a path is determined by its slack and is also shown in the timing analysis report.

Figure 10–31. Critical Paths



To view critical paths in the Timing Closure Floorplan, click the **Critical Path Settings** icon on the toolbar, or, on the View menu, point to Routing and click **Critical Paths Settings**.

When viewing critical paths, you can specify the clock in the design to be viewed. You can determine which paths to display by specifying the slack threshold in the slack field.

Timing settings must be made and timing analysis performed for paths to be displayed in the floorplan.

For more information about performing static timing analyses of your design with a timing analyzer, refer to the *Quartus II Classic Timing Analyzer* and the *Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*.

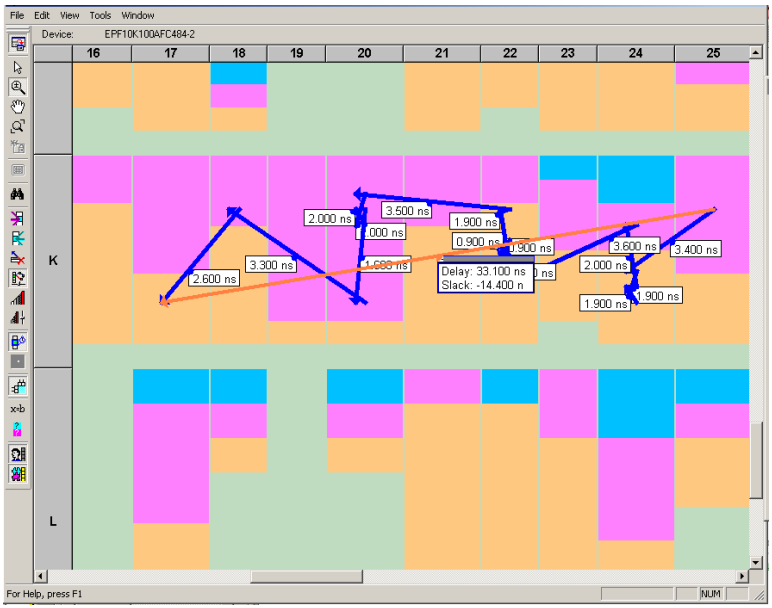
Viewing the critical paths is useful for determining the criticality of nodes based on placement. There are a number of ways to view the details of the critical path.

The default view in the Timing Closure Floorplan shows the path with the source and destination registers displayed. You can also view all the combinational nodes along the worst-case path between the source and destination nodes. To view the full path, click on the delay label to select the path, right-click, and select **Show Path Edges**. Figure 10–32 shows the critical path through combinational nodes. To hide the combinational nodes, select the path, right-click, and select **Hide Path Edges**.



You must view the routing delays to select a path.

Figure 10–32. Worst-Case Combinational Path Showing Path Edges



To assign the path to a LogicLock region using the **Paths** dialog box, select the path, right-click, and select **Properties**.

You can determine the maximum routing delay between two nodes within a LogicLock region. To use this feature, on the View menu, point to Routing and click **Show Intra-region Delay**. Place your cursor over a Fitter-placed LogicLock region to see the maximum delay.

For more information about making path assignments with the **Paths** dialog box, refer to "Timing Closure Floorplan View" on page 10–48.

After running timing analysis, you can locate timing paths from the timing reports file produced. Right-click on any row in the report file, point to Locate, and click **Locate in Timing Closure Floorplan**. The Timing Closure Floorplan window opens with the timing path highlighted.

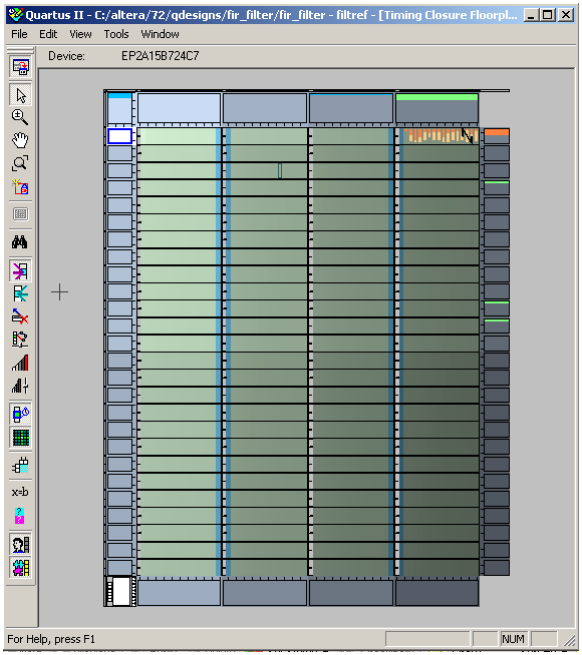


For more information about optimizing your design in the Quartus II software, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*. With the options and tools available in the Timing Closure Floorplan and the techniques described in that chapter, the Quartus II software can help you achieve timing closure in a more time-efficient manner.

Physical Timing Estimates

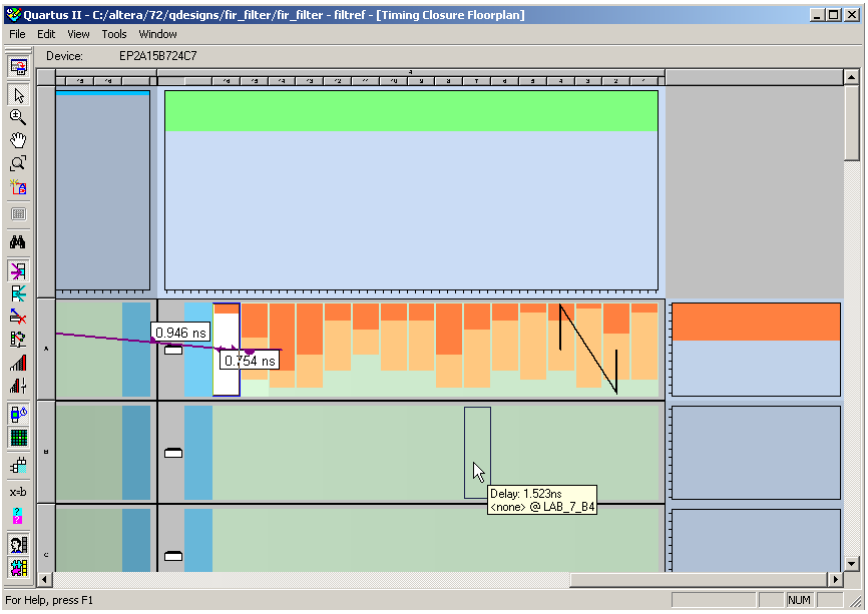
In the Timing Closure Floorplan Editor, you can select a resource and see the approximate delay to any other resource on the device. After you select a resource, the delay is represented by the color of potential destination resources. The darker the color of the resource, the longer the delay (Figure 10–33).

Figure 10–33. Physical Timing Estimates View



You can also obtain an approximation of the delay between two points by selecting a source and holding your cursor over a potential destination resource (Figure 10–34).

Figure 10–34. Delay for Physical Timing Estimate in the Timing Closure Floorplan



The delays represent an estimate based on probable best-case routing. The delay may be greater than what is shown, depending on the availability of routing resources. In general, there is a strong correlation between the probable and actual delay.

To view the physical timing estimates, click the **Show Physical Timing Estimate** icon, or, on the View menu, point to Routing and click **Show Physical Timing Estimates**.

You can use the physical timing estimate information when manually placing logic in a device. This information allows you to place critical nodes and modules closer together, and non-critical or unrelated nodes and modules further apart, reducing the routing congestion between critical and non-critical entities and modules. This placement enables the Quartus II Fitter to meet the timing requirements.

Viewing Routing Congestion

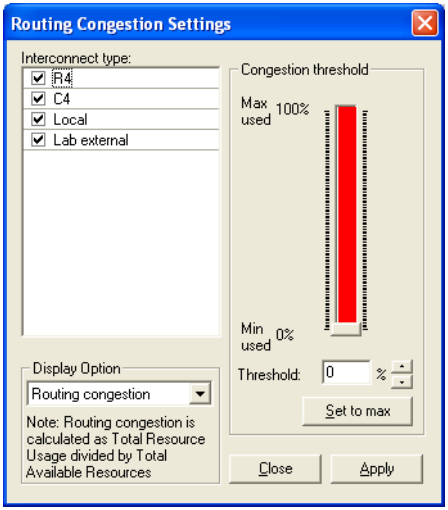
The View Routing Congestion feature allows you to determine the percentage of routing resources used after a compilation. This feature identifies where there is a lack of routing resources.

The congestion is shown by the color and shading of logic resources. The darker shading represents a greater routing resource utilization. Logic resources that are red have routing resource utilization greater than the specified threshold.

The routing congestion view is only available from the View menu when you enable the Field View. To view routing congestion in the floorplan, click the **Show Routing Congestion** icon, or, on the View menu, point to Routing and click **Show Routing Congestion**. To set the criteria for the critical path you want to view, click the **Routing Congestion Settings** icon, or, on the View menu, point to Routing and click **Routing Congestion Settings**.

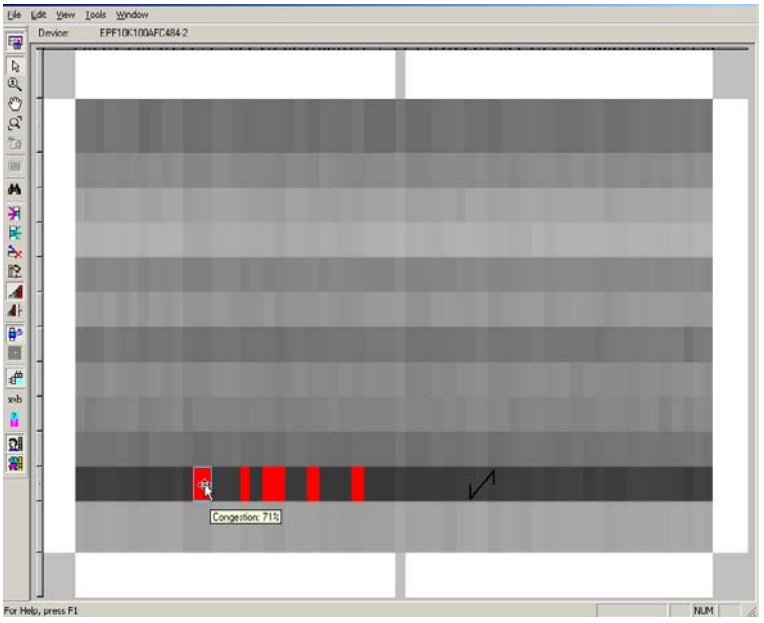
In the **Routing Congestion Settings** dialog box, you can choose the routing resource (interconnect type) you want to examine and set the congestion threshold. Routing congestion is calculated based on the total resource usage divided by the total available resources (Figure 10–35).

Figure 10–35. Routing Congestion Settings Dialog Box



If you use the routing congestion viewer to determine where there is a lack of routing resources, examine each routing resource individually to determine which ones use close to 100% of the available resources (Figure 10–36).

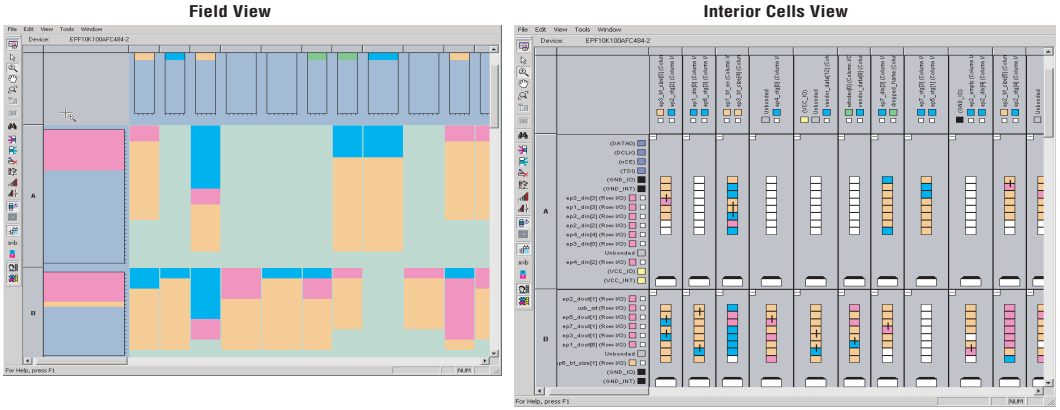
Figure 10–36. Routing Congestion of a Sample Design in a Cyclone Device



Timing Closure Floorplan View

The **Timing Closure Floorplan** view provides you with current and previous compilation assignments on one screen. You can display device resources in either of two views: the Field View and the Interior Cells View, as shown in Figure 10–37. The Field View provides an uncluttered view of the device floorplan in which all device resources, such as embedded system blocks (ESBs) and MegaLAB blocks, are outlined. The Interior Cells View provides a detailed view of device resources, including individual logic elements within a MegaLAB and device pins.

Figure 10–37. Timing Closure Floorplan Editor



LogicLock Regions in the Timing Closure Floorplan

After you have defined a LogicLock region in a device supported by the Timing Closure Floorplan, you must assign resources to it using the Timing Closure Floorplan, the **LogicLock Regions** dialog box, or a Tcl script.

Creating LogicLock Regions in the Timing Closure Floorplan Editor

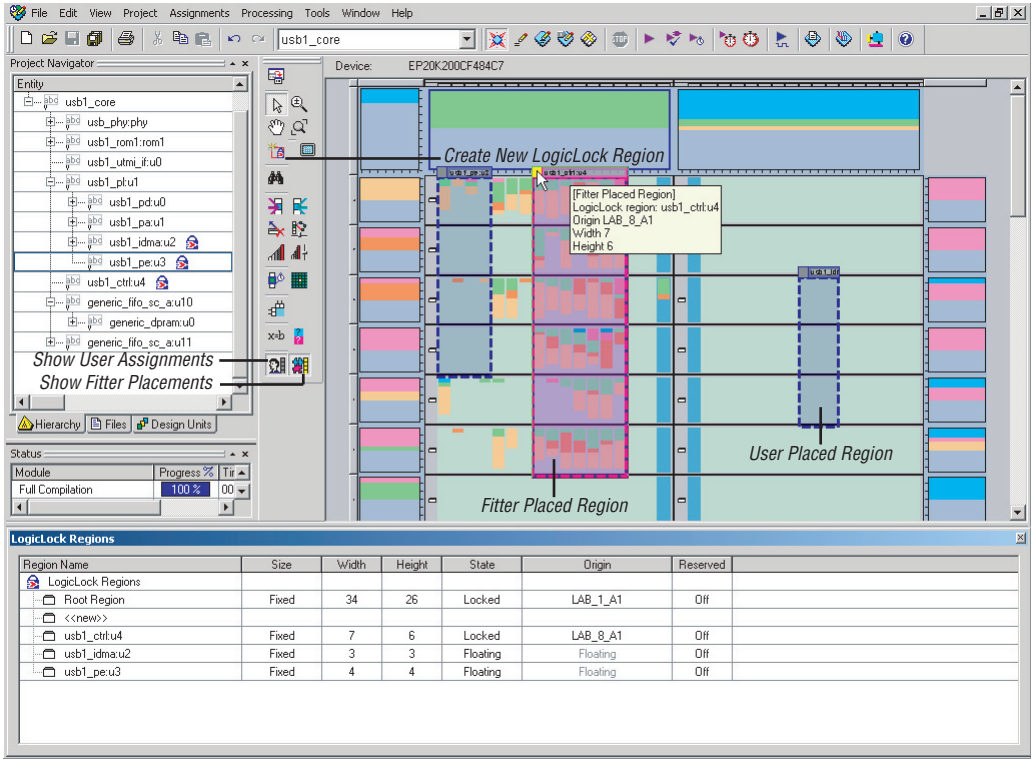
The Timing Closure Floorplan Editor has toolbar buttons that are used to manipulate LogicLock regions as shown in Figure 10–38. You can use the **Create New LogicLock Region** button to draw LogicLock regions in the device floorplan.



The Timing Closure Floorplan Editor displays LogicLock regions when you select **Show User Assignments** or **Show Fitter Placements**. The type of region determines its appearance in the floorplan.

The Timing Closure Floorplan Editor differentiates between user assignments and Fitter placements. When the **Show User Assignments** option is turned on in the Timing Closure Floorplan, you can see current assignments made to a LogicLock region. When the Fitter Placement option is turned on, you can see the properties of the LogicLock region after the most recent compilation. User-assigned LogicLock regions appear in the Timing Closure Floorplan Editor with a dark blue border. Fitter-placed regions appear in the Timing Closure Floorplan Editor with a magenta border (Figure 10–38).

Figure 10–38. Timing Closure Floorplan Editor Toolbar Buttons



Using Drag and Drop to Place Logic

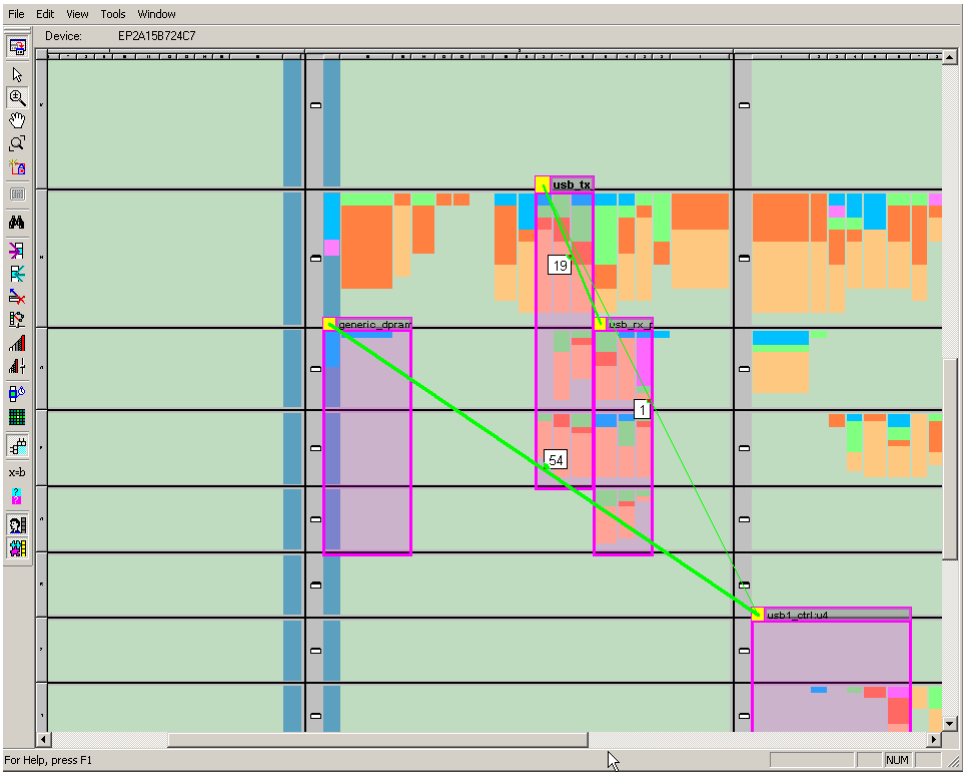
You can drag selected logic displayed in the **Hierarchy** tab of the **Project Navigator**, in the Node Finder, or in a schematic design file, and drop it into the Timing Closure Floorplan or the **LogicLock Regions** dialog box. Figure 10–5 shows logic that has been dragged from the **Hierarchy** tab of the **Project Navigator** and dropped into a LogicLock region in the **Timing Closure Floorplan**.

Analyzing LogicLock Region Connectivity Using the Timing Closure Floorplan

To see how logic in LogicLock regions interfaces, view the connectivity between LogicLock regions. This capability is extremely useful when entities are assigned to LogicLock regions. You can also see the fan-in and fan-out of selected LogicLock regions.

To view the connections in the timing closure floorplan, on the View menu, point to Routing and click **Show LogicLock Regions Connectivity**. Figure 10–39 shows standard LogicLock region connections.

Figure 10–39. LogicLock Region Connections with Connection Count



As shown in Figure 10–39, the thickness of the connection line indicates how many connections exist between regions. To see the number of connections between regions, on the View menu, point to Routing and click **Show Connection Count**.

LogicLock region connectivity is applicable only when the user assignments are enabled in the Timing Closure Floorplan. When you use floating LogicLock regions, the origin of the user-assigned region is not necessarily the same as that of the Fitter-placed region. You can change the origin of your floating LogicLock regions to that of the most recent

compilation either in the LogicLock Regions window or by selecting **Back-Annotate Origin and Lock** under **Location** in the **LogicLock Regions Properties** dialog box.

To view the fan-in or fan-out of a LogicLock region in the Timing Closure Floorplan, select the user-assigned LogicLock region while the fan-in or the fan-out option is turned on.

To turn on the fan-in option, click the **Show Node Fan-In** icon, or, on the View menu, point to Routing and click **Show Node Fan-In**. To turn on the fan-out option, click the **Show Node Fan-Out** icon, or, on the View menu, point to Routing and click **Show Node Fan-Out**. Only the nodes that have user assignments are visible when viewing fan-in or fan-out of LogicLock regions.

Additional Quartus II LogicLock Design Features

To complement the **LogicLock Regions** dialog box and Device Floorplan view, the Quartus II software has additional features to help you design with the LogicLock feature.

Tooltips

When you move the mouse pointer over a LogicLock region name on the **LogicLock Regions** dialog box, or over the top bar of the LogicLock region in the Timing Closure Floorplan, the Quartus II software displays a tooltip with information about the properties of the LogicLock region.

Placing the mouse pointer over Fitter-placed LogicLock regions displays the maximum routing delay within the LogicLock region. To enable this feature, on the View menu, point to Routing and click **Show Intra-region Delay**.

Reserve LogicLock Region

The Quartus II software honors all entity and node assignments to LogicLock regions. Occasionally, entities and nodes do not occupy an entire region, which leaves some of the region's resources unoccupied. To increase the region's resource utilization and performance, the Quartus II software's default behavior fills the unoccupied resources with other nodes and entities that have not been assigned to any other region. You can prevent this behavior by turning on the **Reserved** option on the **General** tab of the **LogicLock Region Properties** dialog box. When this option is turned on, your LogicLock region only contains the entities and nodes that you have specifically assigned to your LogicLock region.

In a team-based design environment, this option is extremely helpful in creating a device floorplan. When this option is turned on, each team can be assigned a portion of the device floorplan where placement and optimization of each submodule occurs. Device resources can be distributed to every module without affecting the performance of other modules.

Prevent Assignment to LogicLock Regions Option

Turning on the **Prevent Assignment to LogicLock Regions** option excludes the specified entity or node from being a member of any LogicLock region. However, it does not prevent the entity or node from physically overlapping LogicLock regions. The Fitter places the entity or node anywhere on the device as if no regions exist. For example, if an entire module is assigned to a LogicLock region, when this option is turned on, you can exclude a specific subentity or node from the region.



You can make the **Prevent Assignment to LogicLock Regions** assignment to an entity or node in the Assignment Editor under **Assignment Name**.

LogicLock Regions Connectivity

The Timing Closure Floorplan Editor allows you to see connections between various LogicLock regions that exist within a design. The connection between the regions is drawn as a single line between the LogicLock regions. The thickness of this line is proportional to the number of connections between the regions.

Rubber Banding

On the View menu, click Routing, and select **Rubber Banding** to show existing connections between LogicLock regions and nodes during movement of LogicLock regions within the Floorplan Editor.

Show Critical Paths

You can display the critical paths in the design by turning on the **Show Critical Paths** option. Use this option with the **Critical Paths Settings** option to display paths based on the Timing Analysis report.

Show Connection Count

You can determine the number of connections between LogicLock regions by turning on the **Show Connection Count** option.

Analysis and Synthesis Resource Utilization by Entity

The Compilation Report contains an **Analysis and Synthesis Resource Utilization by Entity** section, which reports accurate resource usage statistics, including entity-level information. This feature is useful when manually creating LogicLock regions.

Path-Based Assignments

You can assign paths to LogicLock regions based on source and destination nodes, allowing you to easily group critical design nodes into a LogicLock region. Any of the following types of nodes can be the source and destination nodes:

- Valid register-to-register path, meaning that the source and destination nodes must be registers
- Valid pin-to-register path, meaning the source node is a pin and the destination node is a register
- Valid register to pin path, meaning that the source node is a register and the destination node is a pin
- Valid pin-to-pin path, meaning that both the source and destination nodes are pins

To open the **Paths** dialog box, on the **General** tab of the **Logic Lock Regions** dialog box, click **Add Path**.



Both "*" and "?" wildcard characters are allowed for the source and destination nodes. When creating path-based assignments, you can exclude specific nodes using the **Name exclude** field in the **Paths** dialog box. Quartus II ignores all paths passing through the nodes that match the setting in the Name exclude field. For example, consider a case with two paths between the source and destination—one passing through node A and the other passing through node B. If you specify node B in the Name exclude field, only the path assignment through node A is valid.

You can also use the Quartus II Timing Analysis Report to create path-based assignments by following these steps:

1. Expand the **Timing Analyzer** section in the **Compilation Report**.
2. Select any of the clocks in the section labeled "Clock Setup:<*clock name*>."
3. Locate a path that you want to assign to a LogicLock region. Drag this path from the Report window and drop it in the row labeled in the LogicLock Region pane in the Quartus II GUI.

This operation creates a path-based assignment from the source register to the destination register as shown in the **Timing Analysis Report**.

Quartus II Revisions Feature

When you create, modify, or import LogicLock regions into a top-level design, you may need to experiment with different configurations to achieve your desired results. The Quartus II software Revisions feature provides a convenient way to organize the same project with different settings until an optimum configuration is found.

On the Project menu, click **Revisions**. In the **Revisions** dialog box, create and set revisions. Revision can be based on the current design or any previously created revisions. Each Revision may have an associated description. Revisions are a convenient way to organize the placement constraints created for your LogicLock regions.

LogicLock Assignment Precedence

Conflicts can arise during the assignment of entities and nodes to LogicLock regions. For example, an entire top-level entity might be assigned to one region and a node within this top-level entity assigned to another region. To resolve conflicting assignments, the Quartus II software maintains an order of precedence for LogicLock assignments. The following order of precedence, from highest to lowest, applies:

1. Exact node-level assignments
2. Path-based and wildcard assignments
3. Hierarchical assignments

However, conflicts can also occur within path-based and wildcard assignments. Path-based and wildcard assignment conflicts arise when one path-based or wildcard assignment contradicts another path-based or wildcard assignment. For example, a path-based assignment is made containing a node labeled `X` and assigned to LogicLock region `PATH_REGION`. A second assignment is made using wildcard assignment `X*` with node `X` being placed into region `WILDCARD_REGION`. As a result of these two assignments, node `X` is assigned to two regions: `PATH_REGION` and `WILDCARD_REGION`.

To resolve this type of conflict, the Quartus II software maintains the order in which the assignments were made and treats the most recent assignment created with the highest priority.



Open the **Priority** dialog box by selecting **Priority** on the **General** tab of the **LogicLock properties** dialog box. You can change the priority of path-based and wildcard assignments by using the Up and Down buttons in the **Priority** dialog box. To prioritize assignments between regions, you must select multiple LogicLock regions. Once the regions have been selected, you can open the **Priority** dialog box from the LogicLock Properties window.

LogicLock Regions Versus Soft LogicLock Regions

Normally all nodes assigned to a particular LogicLock region reside within the boundaries of that region. Soft LogicLock regions can enhance design performance by removing the fixed rectangular boundaries of LogicLock regions. When you assign the "Soft" property to a LogicLock region, the Quartus II software attempts to place as many nodes assigned to the region as close together as possible, and has the added flexibility to move nodes outside of the soft region to meet your design performance requirement. This process allows the Quartus II Fitter greater flexibility in placing nodes in the device to meet your performance requirements.

When you assign nodes to a soft LogicLock region, they can be placed anywhere in the device, but if the soft region is the child of a region, the nodes are not assigned outside the boundaries of the first non-soft parent region. If a non-soft parent does not exist (in a design targeting a Stratix II, Stratix GX, Stratix, Cyclone II, Cyclone, or MAX II device), the region floats within the `Root_region`, that is, the boundaries of the device. You can turn on the **Soft Region** option on the **Location** tab of the **LogicLock Region Properties** dialog box.



Soft regions can have an arbitrary hierarchy that allows any combination of parent and child to be a soft region. The **Reserved** option is not compatible with soft regions.

Soft LogicLock regions cannot be back-annotated because the Quartus II software may have placed nodes outside of the LogicLock region, resulting in location assignments incompatible with the region's origin and size.

Soft LogicLock regions are available for all device families that support floating LogicLock regions.

Virtual Pins

When you compile a design in the Quartus II software, all I/O ports are directly mapped to pins on the targeted device. The I/O port mapping may create problems for a modular and hierarchical design as lower level

modules may have I/O ports that exceed device pins available on the targeted device. Or, the I/O ports may not directly feed a device pin, but instead drive other internal nodes. The Quartus II software accommodates this situation by supporting virtual pins.

The Virtual Pin assignment communicates to the Quartus II software which I/O ports of the design module are internal nodes in the top-level design. These assignments prevent the number of I/O ports in the lower-level modules from exceeding the total number of available device pins. Every I/O port that is designated a virtual pin is mapped to either an LCELL or an ALM, depending on the target device.



Bidirectional, registered I/O pins, and I/O pins with output enable signals cannot be virtual pins.

In the top-level design, these virtual pins are connected to an internal node of another module. Making assignments to virtual pins allows you to place them in the same location or region on the device as that of the corresponding internal nodes in the top-level module. This feature has the added benefit of providing accurate timing information during lower-level module optimization.

Use the following guidelines for creating virtual pins in the Quartus II software:

- Do not declare clock pins
- Nodes or signals that drive physical device pins in the top-level design should not be declared as virtual pins



In the Node Finder, setting **Filter Type to Pins: Virtual** allows you to display all assigned virtual pins in the design. From the Assignment Editor, to access the Node Finder, double-click the **To** field; when the arrow appears on the right side of the field, click the arrow and select **Node Finder**.

Using LogicLock Methodology for Older Device Families

The LogicLock methodology is recommended as an optimization method only for older device families, such as the MAX II and APEX II device families, that do not support incremental compilation. In this methodology, you optimize your design and lock it down one module at a time. Using the LogicLock feature, you can create and implement each logic module independently in a hierarchical or team-based design. You can use this LogicLock methodology to optimize and preserve timing, placement, or both for devices that do not support incremental compilation.

Using the LogicLock methodology as an optimization strategy is less effective on newer device families such as those device families in the Cyclone and Stratix series of devices. Altera does not recommend using the LogicLock methodology for designs in such devices, although the feature may be supported on some devices in these series. However, you can use LogicLock regions in conjunction with Incremental Compilation to create a floorplan and preserve timing results for these devices in the Stratix and Cyclone series of devices.

For more information about hierarchical and team-based design, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

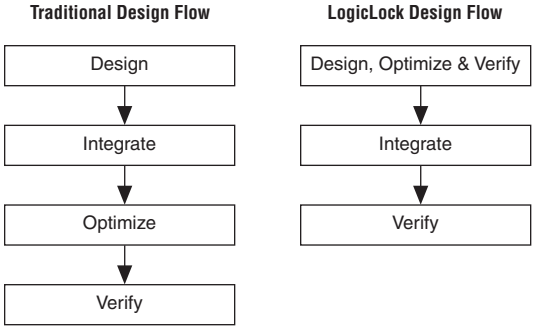
The Quartus II LogicLock Methodology

When you use the LogicLock methodology for older devices, you can place the logic in each netlist file in a fixed or floating region in an Altera device. You can then maintain the placement and, if necessary, the routing of your blocks in the Altera device, thus retaining performance.

To design with the LogicLock feature, create a LogicLock region in a supported device and then assign logic to the region. After you optimize the logic placed within the boundaries of a region to achieve the required performance, you must back-annotate the region's contents to lock the logic placement and routing. Locking the placement and routing preserves the performance when you integrate the region with the rest of the design.

Figures 10–40 compares the traditional design flow with the LogicLock design flow.

Figure 10–40. Traditional Design Flow Compared with Quartus II LogicLock Design Flow





For more information about block-based design with the LogicLock feature, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook.*

Improving Design Performance

The LogicLock methodology helps you optimize and preserve performance. You can use the LogicLock regions to place modules, entities, or any group of logic into regions in a device's floorplan. LogicLock assignments can be hierarchical, which allows you to have more control over the placement and performance of each module and group of modules.

In addition to hierarchical blocks, you can apply LogicLock constraints to individual nodes; for example, you can make a wildcard path-based LogicLock assignment on a critical path. This technique is useful if the critical path spans multiple design blocks.



Although LogicLock constraints can improve performance, they can also degrade performance if they are not applied correctly. They can also result in increased resource usage.

LogicLock Restrictions

During the design process, placing restrictions on nodes or entities in the design is often necessary. These restrictions can conflict with the node or entity assignments for a LogicLock region. To avoid conflicts, consider the order of precedence given to constraints by the Quartus II software during fitting. The following assignments have priority over LogicLock region assignments:

- Assignments to device resources and location assignments
- Fast input-register and fast output-register assignments
- Local clock assignments for Stratix devices
- Custom region assignments
- I/O standard assignments

The Quartus II software removes nodes and entities from LogicLock regions if any of these constraints are applied to them.



After a LogicLock region is back-annotated, the Quartus II software can place the region only in areas of the device with exactly the same resources.

Preserving Timing Results Using the LogicLock Flow

To preserve the timing results for a design module in the Quartus II software, you must preserve the placement and routing information for all the logic in the design module. For older device families, you can use the LogicLock design methodology to back-annotate logic locations within a LogicLock region, which makes assignments to each node in the design.



For more information about block-based design with the LogicLock feature, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

When preserving logic placement in an Altera device using LogicLock back-annotation, an atom netlist preserves the node names in subblocks of your design. An atom netlist contains design information that fully describes the submodule logic in terms of the device architecture. In the atom netlist, the nodes are fixed as Altera primitives and the node names do not change if the atom netlist does not change. If a node name changes, any placement information associated with that node, such as LogicLock assignments made when back-annotating a region, is invalid and ignored by the compiler.

If all the netlists are contained in one Quartus II project, use the LogicLock flow to back-annotate the logic in each region. If a design region changes, only the netlist associated with the changed region is affected. When you place and route the design using the Quartus II software, the software needs to re-fit only the LogicLock region associated with the changed netlist file.



Turn on the **Prevent further netlist optimization** option when back-annotating a region with the **Synthesis Netlist Optimizations** option, the **Physical Synthesis Optimization** option, or both, turned on. This sets the **Netlist Optimizations** option to **Never Allow** for all nodes in the region, preventing node name changes in the top-level design when the region is recompiled.

You must remove previously back-annotated assignments for a modified block if the node names are different in the newly synthesized version. When you recompile with one new netlist file, the placement and assignments for the unchanged netlist files assigned to other LogicLock regions are not affected. Therefore, you can make changes to code in an independent block and not interfere with another designer's changes, even when all the blocks are integrated into the same top-level design.

With the LogicLock design methodology, you can develop and test submodules without affecting other areas of a design.

Importing and Exporting LogicLock Regions

This section describes the steps required to import and export the LogicLock regions when you are using the LogicLock methodology as an optimization tool for older devices.



For information about importing and exporting the assignments for lower-level design partitions using the incremental compilation flow, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

For the Quartus II software to achieve optimal placement, you must specify timing assignments, including t_{SU}, t_{CO}, and t_{PD}, for all clock signals in the design.

To facilitate the LogicLock design flow, the **Timing Closure Floorplan** highlights resources that have back-annotated LogicLock regions.

Export the Module

This section describes how to export a module's constraints in a format that can be imported to a top-level design. To be exported, a module requires design information stored as an atom netlist (VQM or EDF), placement information stored in a Quartus II Settings File, and routing information stored in a Routing Constraints File (**.rcf**).

Atom Netlist Design Information
The atom netlist contains design information that fully describes the module's logic in terms of an Altera device architecture. If the design was synthesized using a third-party tool and then brought into the Quartus II software, an atom netlist already exists and the node names are fixed. You do not have to generate another atom netlist. However, if you use any synthesis netlist optimizations or physical synthesis optimizations, you must generate a Verilog Quartus Mapping Netlist File (**.vqm**) using the Quartus II software, because the original atom netlist may have changed as a result of these optimizations.



Turn on the **Prevent further netlist optimization** option when back-annotating a region with the **Synthesis Netlist Optimizations** and/or **Physical Synthesis Optimization** options turned on. This sets the **Netlist Optimizations** to **Never Allow** for all nodes in the region, avoiding the possibility of a node name change when the region is imported into the top-level design.

If you synthesized the design from a VHDL Design File (**.vhd**), Verilog Design File (**.v**), Text Design File (**.tdf**), or Block Design File (**.bdf**) in the Quartus II software, you must also create an atom netlist to fix the node names. During compilation, the Quartus II software creates a Verilog Quartus Mapping Netlist File in the **atom_netlists** subdirectory in the project directory.



If the atom netlist is generated by a third-party synthesis tool and the design has a black box library of parameterized modules (LPM) functions or Altera megafunctions, you must generate a Quartus II Verilog Quartus Mapping Netlist File for the black box modules.



For instructions about creating an atom netlist in the Quartus II software, refer to *Saving Synthesis Results to a Verilog Quartus Mapping File* in the Quartus II Help.

When you export LogicLock regions, all your design assignments are exported. Filtering occurs only when the design is imported. However, you can export a subentity of the compilation hierarchy and all of its relevant regions. To do this, right-click on the entity in the **Hierarchy** tab of the **Project Navigator** and click **Export Assignments.**

Placement Information
The Quartus II Settings File contains the module's LogicLock constraint information, including clock settings, pin assignments, and relative placement information for back-annotated regions. To maintain performance, you must back-annotate the module.

Routing Information
The Routing Constraints File (**.rcf**) contains the module's LogicLock routing information. To maintain performance, you must back-annotate the module.

Exporting the Routing Constraint File and Atom Netlist
To specify the Routing Constraint File and Atom Netlist to export, perform the following steps:

1. Run a full compilation.
2. On the Assignments menu, click **LogicLock Regions Window**.
3. Right-click the region name, and click **Properties**.

4. In the **LogicLock Region Properties** dialog box, click **Back-Annotate Contents**.
5. Enable or disable any of the advanced options such as **Prevent further netlist optimization**.
6. Turn on **Routing**, and click **OK**.
7. In the **LogicLock Region Properties** dialog box, click **OK**.
8. On the Assignments menu, click **Export Assignments**.
9. In the **Export Assignments** dialog box, turn on **Export back-annotated routing** and **Save a node-level netlist of the entire design into a persistent source file**, then click **OK**.



For instructions about exporting a LogicLock region assignment in the Quartus II software, refer to *Importing and Exporting LogicLock Region Assignments* in the Quartus II Help.

Import the Module

To specify the Quartus II Settings File for a specific instance or entity, use the **LogicLock Import File Name** option in the Assignment Editor. This option lets you specify different LogicLock region constraints for each instance of an entity and import them into the top-level design. You can also specify an RCF file with the **LogicLock Routing Constraints File Name** option in the Assignment Editor.

When importing LogicLock regions into the top-level design, you must specify the Quartus II Settings File and Routing Constraints File for the modules in the project. If the design instantiates a module multiple times, the Quartus II software applies the LogicLock regions multiple times.



Before importing LogicLock regions, you must perform analysis and elaboration, or compile the top-level design, to ensure the Quartus II software is aware of all instances of the lower-level modules.

The following sections describe how to specify a Quartus II Settings File for a module and how to import the LogicLock assignments into the top-level design.

Importing the Routing Constraints File and the Atom Netlist File
To specify the Quartus II Settings File and atom netlist to import, perform the following steps:

1. On the Assignments menu, click **Import Assignments**. In the **Import Assignments** dialog box, click **Advanced**.

2. In the **Advanced Import Settings** dialog box, turn on **Back-annotated routing**.

Now, when you import a LogicLock region, the routing constraint file is also imported.

Import the Assignments
On the Assignments menu, click **Import Assignments** to import the assignments. The **Import Assignments** dialog box appears. In the **Import Assignments** dialog box, click **Advanced**. The **Advanced Import Settings** dialog box appears. Use the options available in the **Advanced Import Settings** dialog box to control how you import your LogicLock regions.

To prevent spurious no-fit errors, parent or top-level regions with multiple instances (that do not contain back-annotated routing information) are imported with their states set to floating. Otherwise, the region's state remains as specified in the Quartus II Settings File. This default allows the Quartus II software to move LogicLock regions to areas on the device with free resources. A child region is locked or floating relative to its parent region's origin as specified in the module's original LogicLock constraints.



If you want to lock a LogicLock region to a location, you can manually lock down the region in the **LogicLock Regions** dialog box or the Timing Closure Floorplan.

Each imported LogicLock region has a name that corresponds to the original LogicLock region name combined with the instance name in the form of <*instance name*>|<*original LogicLock region name*>. For example, if a LogicLock region for a module is named `LLR_0` and the instance name for the module is `Filter:inst1`, then the LogicLock region name in the top-level design is `Filter:inst1|LLR_0`.

Compile and Verify the Top-Level Design

After importing all modules, you can compile and verify the top-level design. The compilation report shows whether system timing requirements are met.

Back-Annotating Routing Information

LogicLock regions not only allow you to preserve the placement of logic from one compilation to the next, they also allow you to retain the routing inside the LogicLock regions. With both placement and routing locked, you have an extremely portable design module that can be used many times in a top-level design without requiring further optimization.



Back-annotate routing only when necessary because doing so can prevent the Quartus II Fitter from finding an optimal fit for your design.

Back-annotate the routing from the Assignments menu, by choosing **Routing** from the **Back-Annotate Assignments** dialog box.



If you are not using an atom netlist, you must turn on the **Save a node-level netlist of the entire design into a persistent source file** option (on the Assignments menu, click **Back-Annotate Assignments**) if back-annotation of routing is selected. Writing out a Verilog Quartus Mapping Netlist File causes the Quartus II software to enforce persistent naming of nodes when saving the routing information. The Verilog Quartus Mapping Netlist File is then used as the design's source.

Back-annotated routing information is valid only for regions with fixed sizes and locked locations. The Quartus II software ignores the routing information for LogicLock regions you specify as floating and automatically sized.

The **Disable Back-Annotated Node locations** option in the **LogicLock Region Properties** dialog box is not available if the region contains both back-annotated routing and back-annotated nodes.

Back-Annotating LogicLock Regions

To back-annotate the contents of your LogicLock regions, perform these steps:

1. In the **LogicLock Region Properties** dialog box, click **Back-Annotate Contents**. The **Back-Annotate Assignments** dialog is shown.
2. In the **Back-Annotate Assignments** dialog box, in the **Back annotation type** list, select **Advanced** and click **OK**.
3. In the **LogicLock Region Properties** dialog box, click **OK**.



If you are using the incremental compilation flow, logic back-annotation is not required. Preserve placement results using the Post-Fit Netlist Type instead of making placement assignments with back-annotation as described in this section.



You can also back-annotate routing within LogicLock regions to preserve performance of the regions. For more information about back-annotating routing, refer to "Back-Annotating Routing Information" on page 10–65.

When you back-annotate a region's contents, all of the design element nodes appear under **Back-annotated nodes** with an assignment to a device resource under **Node Location**, for example, logic array block (LAB), memory block, or digital signal processing (DSP) block. Each node's location is the placement of the node after the most recent compilation. If the origin of the region changes, the node's location changes to maintain the same relative placement. This relative placement preserves the performance of the module. If cell assignments are demoted, the nodes are assigned to LABs rather than directly to logic cells. This provides more flexibility to the Fitter, and improves the chances of a fit.

Exporting Back-Annotated Routing in LogicLock Regions

To export the LogicLock region routing information, on the Assignments menu, click **Export Assignments,** and in the **Export Assignments** dialog box, turn on **Export Back-annotated routing**. This option generates a Quartus II Settings File and a Routing Constraints File in the specified directory. The Quartus II Settings File contains all LogicLock region properties as specified in the current design. The Routing Constraints File contains all the necessary routing information for the exported LogicLock regions.

This Routing Constraints File works only with the atom netlist for the entity being exported.

Only regions that have back-annotated routing information have their routing information exported when you export the LogicLock regions. All other regions are exported as regular LogicLock regions.

To determine if a LogicLock region contains back-annotated routing, refer to the **Content Status** box shown on the **Contents** tab of the **LogicLock Region Properties** dialog box. If routing has been back-annotated, the status is "Nodes and Routing Back-Annotated".

The Quartus II software also reports whether routing information has been back-annotated in the Timing Closure Floorplan. LogicLock regions with back-annotated routing have an "R" in the top-left corner of the region.

Importing Back-Annotated Routing in LogicLock Regions

To import LogicLock region routing information, you must specify the instance that will have its routing information imported. This is done with the assignment LogicLock Routing Constraints File in the Assignment Editor.



A Routing Constraints File must be explicitly specified using the LogicLock **Back-annotated Routing Import File Name** assignment before importing any LogicLock region.

The Quartus II software imports LogicLock regions with back-annotated routing as regions locked to a location and of fixed size.

You can import back-annotated routing if only one instance of the imported region exists in the top level of the design. If more than one instance of the imported region exists in the top level of the design, the routing constraint is ignored and the LogicLock region is imported without back-annotation of routing. This is because routing resources from one part of the device may not be exactly the same in another area of the device.



When importing the Routing Constraints File for a lower level entity, you must use the same atom netlist, that is, the Verilog Quartus Mapping Netlist File that was used to generate the Routing Constraints File. This restriction ensures that the node names annotated in the Routing Constraints File match those in the atom netlist.

Scripting Support

You can run procedures and create the settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```



The same information is available in the Quartus II Help, and in PDF format in the *Quartus II Scripting Reference Manual*.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

for information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Reference Manual*.

Initializing and Uninitializing a LogicLock Region

You must initialize the LogicLock data structures before creating or modifying any LogicLock regions and before executing any of the Tcl commands listed below.

Use the following Tcl command to initialize the LogicLock data structures:

```
initialize_logiclock
```

Use the following command to uninitialize the LogicLock data structures before closing your project:

```
uninitialize_logiclock
```

Creating or Modifying LogicLock Regions

Use the following Tcl command to create or modify a LogicLock region:

```
set_logiclock -auto_size true -floating true -region \
<my_region-name>
```

In the above example, the region's size is set to auto and the state set to floating.

If you specify a region name that does not exist in the design, the command creates the region with the specified properties. If you specify the name of an existing region, the command changes all properties you specify, and leaves unspecified properties unchanged.

For more information about creating LogicLock regions, refer to the sections "Creating LogicLock Regions" on page 10–7 and "Creating LogicLock Regions with the Chip Planner" on page 10–12.

Obtaining LogicLock Region Properties

Use the following Tcl command to obtain LogicLock region properties. This example returns the height of the region named `my_region`.

```
get_logiclock -region my_region -height
```

Assigning LogicLock Region Content

Use the following Tcl commands to assign or change nodes and entities in a LogicLock region. This example assigns all nodes with names matching `fifo*` to the region named `my_region`.

```
set_logiclock_contents -region my_region -to fifo*
```

You can also make path-based assignments with the following Tcl command:

```
set_logiclock_contents -region my_region -from \
fifo -to ram*
```

For more information about assigning LogicLock Region Content, refer to "Assigning LogicLock Region Content" on page 10–11.

Prevent Further Netlist Optimization

Use this Tcl code to prevent further netlist optimization for nodes in a back-annotated LogicLock region. In your code, specify the name of your LogicLock region.

```
foreach node [get_logiclock_contents -region \
<region name> -node_locations] {

    set node_name [lindex $node 0]

    set_instance_assignment -name
ADV_NETLIST_OPT_ALLOWED "NEVER ALLOW" -to $node_name
```

The `get_logiclock_contents` command is in the `logiclock` package.

Save a Node-Level Netlist for the Entire Design into a Persistent Source File

Make the following assignments to cause the Quartus II Fitter to save a node-level netlist for the entire design into a Verilog Quartus Mapping Netlist File:

```
set_global_assignment \
-name LOGICLOCK_INCREMENTAL_COMPILE_ASSIGNMENT ON
set_global_assignment \
-name LOGICLOCK_INCREMENTAL_COMPILE_FILE <file name>
```

Any path specified in the file name is relative to the project directory. For example, specifying **atom_netlists/top.vqm** places **top.vqm** in the **atom_netlists** subdirectory of your project directory.

A Verilog Quartus Mapping Netlist File is saved in the directory specified at the completion of a full compilation.

For more information about saving a node-level netlist, refer to "Atom Netlist Design Information" on page 10–61.

Exporting LogicLock Regions

Use the following Tcl command to export LogicLock region assignments. This example exports all LogicLock regions in your design to a file called **export.qsf**.

```
logiclock_export -file export.qsf
```

For more information about exporting LogicLock regions refer to "Export the Module" on page 10–61.

Importing LogicLock Regions

Use the following Tcl commands to import LogicLock region assignments. This example ignores any pin assignments in the imported region.

```
set_instance_assignment -name LL_IMPORT_FILE \
my_region.qsf -to my_destination

logiclock_import -no_pins
```

Running the import command imports the assignment types for each entity in the design hierarchy. The assignments are imported from the file specified in the `LL_IMPORT_FILE` setting.

For more information about importing LogicLock regions, refer to "Import the Module" on page 10–63.

Setting LogicLock Assignment Priority

Use the following Tcl code to set the priority for a LogicLock region's members. This example reverses the priorities of the LogicLock region in your design.

```
set reverse [list]
for each member [get_logiclock_member_priority] {
    set reverse [insert $reverse 0 $member]
{
set_logiclock_member_priority $reverse
```

For more information about setting the LogicLock assignment priority, refer to "LogicLock Restrictions" on page 10–59.

Assigning Virtual Pins

Use the following Tcl command to turn on the virtual pin setting for a pin called `my_pin`:

```
set_instance_assignment -name VIRTUAL_PIN ON \
-to my_pin
```

For more information about assigning virtual pins, refer to "Virtual Pins" on page 10–56.

Back-Annotating LogicLock Regions

The Quartus II software provides the back-annotate Tcl package that allows you to back-annotate the contents of a LogicLock region.

```
logiclock_back_annotate [-h | -help] [-long_help]
[-region <region name>] [-from <source name>]
[-to <destination name>] [-exclude_from] [-exclude_to] [-path_exclude <path_exclude name>]
[-no_delay_chain] [-no_contents] [-lock] [-routing]
[-resource_filter <resource_filter value>] [-no_dont_touch]

[-remove_assignments] [-no_demote_lab] [-no_demote_mac] [-no_demote_pin] [-no_demote_ram]
```

For example, the following command back-annotates all nodes and routing in the region, `one_region`.

```
package require ::quartus::backannotate
logiclock_back_annotate -routing -lock -no_demote_lab -region one_region
```



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Conclusion

Design analysis for timing closure is a fundamental requirement for optimal performance in highly complex designs. With their analysis capability, the Quartus II Timing Closure Floorplan Editor and Chip Planner tools help you close timing quickly on complex designs. Using these tools along with the LogicLock and Incremental Compilation methodologies, enables you to compile your designs hierarchically, preserving the timing results from individual compilation runs. You can use LogicLock regions as part of an incremental compilation methodology to improve your productivity. You can also include a module in one or more projects while maintaining performance and reducing development costs and time-to-market. LogicLock region assignments give you complete control over logic and memory placement, and you can use LogicLock region assignments to improve the performance of non-hierarchical designs as well.

Referenced Documents

This chapter references the following documents:

- *AN 437: Power Optimization in Stratix III FPGAs*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Reference Manual*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 10–3 shows the revision history for this chapter.

Table 10–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Updated for the Quartus II 7.2 software release, including: • Reorganization of sections to emphasize the Chip Planner. • Updated instructions.	Updated for the Quartus II 7.2 software release.
May 2007 v7.1.0	This is a new chapter, consisting of material from the following *Quartus II 7.0 Handbook* chapters: • *Timing Closure Floorplan* (volume 2) • *LogicLock Design Methodology* (volume 3) • *Design Analysis and Engineering Change Management with the Chip Planner* (volume 3) The following changes were also made: • All of the information about Floorplan Analysis in these chapters has been integrated into this new chapter. All ECO-related sections have moved to a different chapter. • Updated several screenshots. • Updated device support. • Added Referenced Documents section.	Updated for the Quartus II 7.1 software release.

11. Netlist Optimizations and Physical Synthesis

QII52007-7.2.0

Introduction

The Quartus® II software offers advanced netlist optimization options, including physical synthesis, to optimize your design beyond the optimization performed in the course of the standard Quartus II compilation flow. The effect of these options depends on the structure of your design, but netlist optimizations can help improve the performance of your design regardless of the synthesis tool used. Device support for these optimizations varies; see the appropriate section for details.

Netlist optimization options work with your design's atom netlist, which describes a design in terms of Altera®-specific primitives. An atom netlist file can take the form of an Electronic Design Interchange Format file (**.edf**) or a Verilog Quartus Mapping file (**.vqm**) generated by a third-party synthesis tool, or a netlist used internally by the Quartus II software. Netlist optimizations are applied at different stages of the Quartus II compilation flow, either during synthesis or during fitting.

The synthesis netlist optimizations occur during the synthesis stage of the Quartus II compilation flow. The synthesis netlist optimizations make changes to the synthesis netlist output from a third-party synthesis tool or make changes as an intermediate step in Quartus II integrated synthesis (one of the optimizations applies only to third-party synthesis netlists). These netlist changes are beneficial in terms of area or speed, depending on your selected optimization technique.

Physical synthesis optimizations take place during the Fitter stage of the Quartus II compilation flow. These optimizations make placement-specific changes to the netlist that improve performance results for a specific Altera device.

This chapter explains how the netlist optimizations in the Quartus II software can modify your design's netlist and help improve your quality of results. The sections "Synthesis Netlist Optimizations" on page 11–3 and "Physical Synthesis Optimizations" on page 11–11 explain how the available optimizations work. This chapter also provides information about preserving your compilation results through back-annotation and writing out a new netlist, and provides guidelines for applying the various options.



When synthesis netlist optimization or physical synthesis options are turned on, the node names for primitives in the design can change. The fact that nodes may be renamed must be considered if you are using a LogicLock™ or verification flow that may require fixed node names, such as the SignalTap® II logic analyzer or formal verification. If your design flow requires fixed node names, you may need to turn off the synthesis netlist optimization and physical synthesis options.

Primitive node names are specified during synthesis. When netlist optimizations are applied, node names may change as primitives are created and removed. Hardware description language (HDL) attributes applied to preserve logic in third-party synthesis tools cannot be honored because those attributes are not written into the atom netlist read by the Quartus II software. If you are synthesizing in the Quartus II software, you can use the Preserve Register (`preserve`) and Keep Combinational Logic (`keep`) attributes to maintain certain nodes in the design.

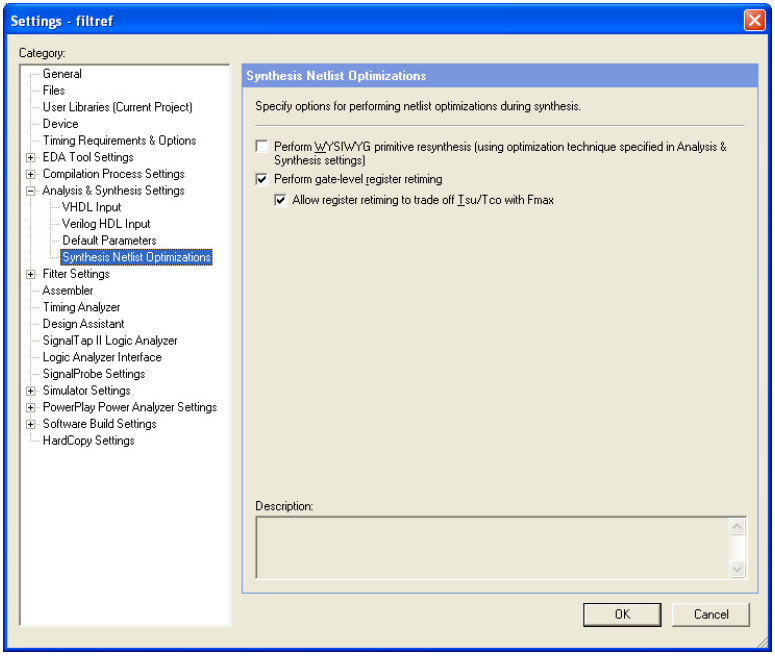


For more information about using these attributes during synthesis in the Quartus II software, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook.*

Synthesis Netlist Optimizations

To view and modify the synthesis netlist optimization options, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, select **Synthesis Netlist Optimizations**, and specify the options for performing netlist optimization during synthesis, as shown in Figure 11–1.

Figure 11–1. Synthesis Netlist Optimizations Page



The sections "WYSIWYG Primitive Resynthesis" and "Gate-Level Register Retiming" on page 11–5 describe these synthesis netlist optimizations, and how they can help improve the quality of results for your design.

WYSIWYG Primitive Resynthesis

You can use the **Perform WYSIWYG primitive resynthesis (using optimization technique specified in Analysis & Synthesis settings)** synthesis option when you have an atom netlist file that specifies a design as Altera-specific primitives. Atom netlist files can take the form of either an Electronic Design Interchange Format file or a Verilog Quartus Mapping file generated by a third-party synthesis tool. To select this option, on the Assignments menu, click **Settings**. In the **Category list**, select **Analysis & Synthesis Settings**, select **Synthesis Netlist**

Optimizations, and turn on **Perform WYSIWYG primitive resynthesis (using optimization technique specified in Analysis & Synthesis Settings)**. If you want to perform WYSIWYG resynthesis on only a portion of your design, you can use the Assignment Editor to assign the **Perform WYSIWYG primitive resynthesis** logic option to a lower-level entity in your design. This option can be used with Arria™ GX, HardCopy® series, Stratix® series, Cyclone® series, MAX® II, or APEX™ series device families.

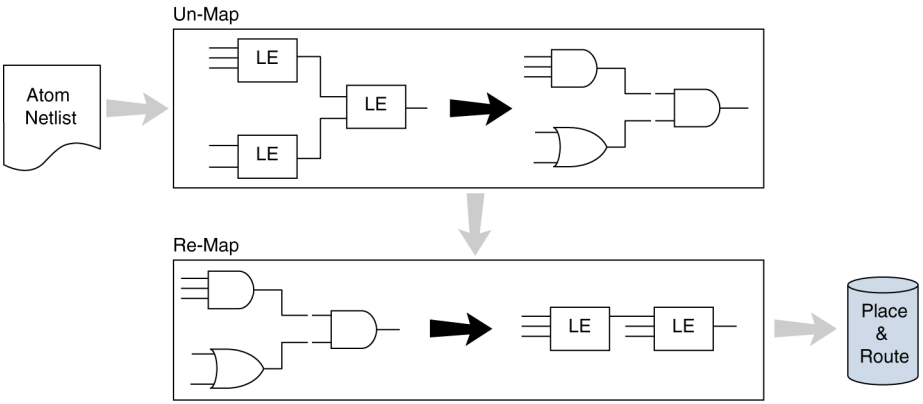
The Perform WYSIWYG primitive resynthesis option directs the Quartus II software to un-map the logic elements (LEs) in an atom netlist to logic gates, and then re-map the gates back to Altera-specific primitives. This feature allows the Quartus II software to use different techniques specific to the device architecture during the re-mapping process. This feature re-maps the design using the Optimization Technique specified for your project.

To turn on this option, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**. In the **Analysis & Synthesis Settings** page, under **Optimization Technique**, select **Speed**, **Area**, or **Balanced** to specify how the Quartus II technology mapper optimizes the design. The **Balanced** setting is the default for many Altera device families; this setting optimizes the timing critical parts of the design for speed and the rest for area.

Refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook* for details on the Optimization Technique option.

Figure 11–2 shows the Quartus II software flow for this feature.

Figure 11–2. WYSIWYG Primitive Resynthesis



The **Perform WYSIWYG primitive resynthesis** option is not applicable if you are using Quartus II integrated synthesis. With the Quartus II synthesis, you do not have to un-map Altera primitives; they are already mapped during the synthesis step using the techniques that are used with the WYSIWYG primitive resynthesis option.

The **Perform WYSIWYG primitive resynthesis** option un-maps and re-maps only logic cell, also referred to as LCELL or LE primitives, and regular I/O primitives (which may contain registers). Double data rate (DDR) I/O primitives, memory primitives, digital signal processing (DSP) primitives, and logic cells in carry/cascade chains are not touched. Logic specified in an encrypted Verilog Quartus Mapping file or an Electronic Design Interchange Format file, such as third-party intellectual property (IP), is not touched.

Turning on this option can cause drastic changes to the node names in the Verilog Quartus Mapping file or Electronic Design Interchange Format file from your third-party synthesis tool, because the primitives in the atom netlist are being broken apart and then remapped within the Quartus II software. Registers can be minimized away and duplicates removed, but registers that are not removed have the same name after remapping.

Any nodes or entities that have the **Netlist Optimizations** logic option set to **Never Allow** are not affected during WYSIWYG primitive resynthesis. To apply this logic option, on the Assignments menu, click **Assignment Editor**. This option disables WYSIWYG resynthesis for parts of your design.

Gate-Level Register Retiming

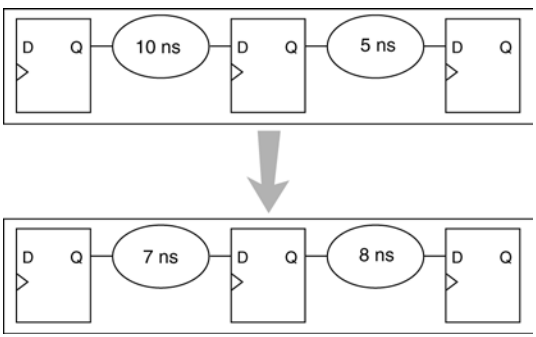
The **Perform gate-level register retiming** option enables movement of registers across combinational logic to balance timing, allowing the Quartus II software to trade off the delay between timing-critical paths and non-critical paths. See Figure 11–3 on page 11–6 for an example. This option can be used with Arria GX, HardCopy series, Stratix series, Cyclone series, MAX II, and APEX series device families. To set this option, on the Assignments menu, click **Settings**. In the **Category list**, select **Analysis & Synthesis Settings**, select **Synthesis Netlist Optimizations**. In the **Synthesis Netlist Optimizations** page, turn on **Perform gate-level register retiming**.

The functionality of your design is not changed when the **Perform gate-level register retiming** option is turned on. However, if any registers in your design have the **Power-Up Don't Care** logic option assigned, the values of registers during power-up may change due to this register and logic movement. The **Power-Up Don't Care** logic option is turned on globally by default. To change the default setting for this option, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**. In the **Analysis & Synthesis Settings** page, click **More Settings**.

You can set the **Power-Up Don't Care** logic option for individual registers or entities using the Assignment Editor. You can also specify a power-up value for individual registers or entities with the **Power-Up Level** logic option. Registers that are explicitly assigned power-up values are not combined with registers that have been explicitly assigned other values.

Figure 11–3 shows an example of gate-level register retiming where the 10 ns critical delay is reduced by moving the register relative to the combinational logic.

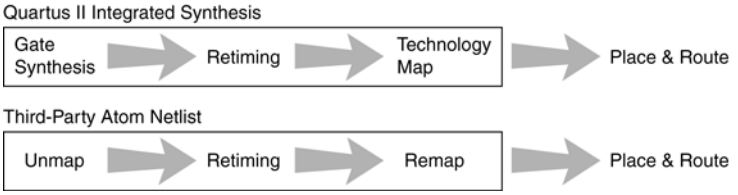
Figure 11–3. Gate-Level Register Retiming Diagram



Register retiming makes changes at the gate level. If you are using an atom netlist from a third-party synthesis tool, you must also use the **Perform WYSIWYG primitive resynthesis** option to un-map atom primitives to gates (so that register retiming can be performed) and then to re-map gates to Altera primitives. If your design uses Quartus II integrated synthesis, retiming occurs during synthesis before the design is mapped to Altera primitives. Megafunctions instantiated in a design are always synthesized using the Quartus II software.

The design flows for the case of integrated Quartus II synthesis and a third-party atom netlist are shown in Figure 11–4.

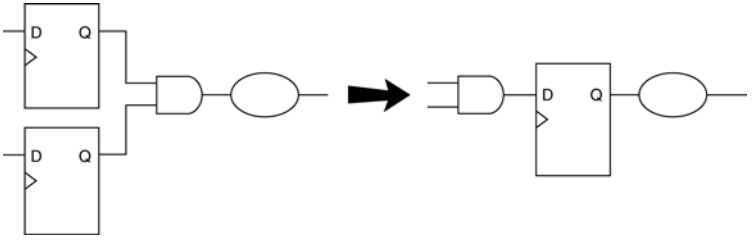
Figure 11–4. Gate-Level Synthesis



The gate-level register retiming option only moves registers across combinational gates. Registers are not moved across LCELL primitives instantiated by the user, memory blocks, DSP blocks, or carry/cascade chains that you have instantiated. Carry/cascade chains are always left intact when performing register retiming.

One benefit of register retiming is the ability to move registers from the inputs of a combinational logic block to the output, potentially combining the registers. In this case, some registers are removed, and one is created at the output, as shown in Figure 11–5.

Figure 11–5. Combining Registers with Register Retiming



The register retiming option can only move and combine registers in this type of situation if the following conditions are met:

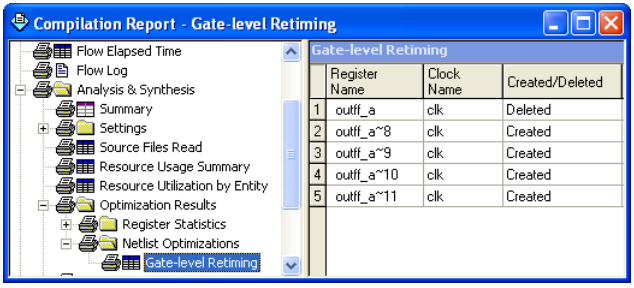
- All registers have the same clock signal
- All registers have the same clock enable signal
- All registers have asynchronous control signals that are active under the same conditions
- Only one register has an asynchronous load other than VCC or GND

Retiming can always create multiple registers at the input of a combinational block from a register at the output of a combinational block. In this case, the new registers have the same clock and clock enable. The asynchronous control signals and power-up level are derived from previous registers to provide equivalent functionality.

The **Gate-level Retiming** report provides a list of registers that were created and removed during register retiming. To access this report, on the Processing menu, click **Compilation Report**. In the **Analysis & Synthesis** list, select **Optimization Results**, select **Netlist Optimizations**, and click **Gate-level Retiming** (Figure 11–6).

The node names for these registers change during the retiming process.

Figure 11–6. Gate-Level Retiming Report



You can set the **Netlist Optimizations** logic option to **Never Allow** to prevent register movement during register retiming. This option can be applied either to individual registers or entities in the design using the Assignment Editor.

The following registers are not moved during gate-level register retiming:

- Registers that have any timing constraint other than global f_{MAX}, t_{SU}, or t_{CO}. For example, any node affected by a Multicycle or Cut Timing assignment is not moved.
- Registers that feed asynchronous control signals on another register.
- Registers feeding the clock of another register.
- Registers feeding a register in another clock domain.
- Registers that are fed by a register in another clock domain.
- Registers connected to serializer/deserializer (SERDES).
- Registers that have the **Netlist Optimizations** logic option set to **Never Allow**.

- Registers feeding output pins (without logic between the register and the pin).
- Registers fed by an input pin (without logic between register and input pin).
- Both registers in a direct connection from input pin-to-register-to-register if both registers have the same clock and the first register does not fan out to anywhere else. These registers are considered synchronization registers.
- Both registers in a direct connection from register-to-register if both registers have the same clock, the first register does not fan out to anywhere else, and the first register is fed by another register in a different clock domain (directly or through combinational logic). These registers are considered synchronization registers.

You can change the retiming behavior for a sequence of synchronization or meta-stability registers by changing the value of the **Retiming Meta-Stability Register Sequence Length** logic option. The value of this option indicates the number of synchronization registers that will not be moved during gate-level register retiming. The default value is 2. To set the value to any number greater than 0, on the Assignments menu, click Settings. In the Settings dialog box, select **Analysis & Synthesis Settings** and click **More Settings**. A value of 1 means that any registers connected to the first register in a register-to-register connection can be moved during retiming. A value of $n > 1$ means that any registers in a sequence of length 1, 2,… n are not moved during gate-level register retiming as long as all of the following are true:

- The first register is fed either directly by a pin or by a register in another clock domain (directly or through combinational logic)
- All registers in the sequence have the same clock
- All but the last register feed the next register in the sequence directly and do not fan out to anywhere else

If you want to consider registers with any of these conditions for register retiming, you can override these rules by setting the **Netlist Optimizations** logic option to **Always Allow** for a given set of registers.

Allow Register Retiming to Trade-Off t_{SU}/t_{CO} with f_{MAX}

To determine whether the Quartus II compiler should attempt to increase f_{MAX} at the expense of t_{SU} or t_{CO} times, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and select **Synthesis Netlist Optimizations**. In the **Synthesis Netlist Optimizations** page, turn on **Allow register retiming to trade off Tsu/Tco with Fmax**. This option affects the optimizations performed due to the gate-level register retiming option.

When both the **Perform gate-level register retiming** and the **Allow register retiming to trade off Tsu/Tco with Fmax** options are turned on, retiming can affect registers that feed and are fed by I/O pins. If the latter option is not turned on, the retiming option does not touch any registers that connect to I/O pins through one or more levels of combinational logic.

Preserving Synthesis Netlist Optimization Results

The Quartus II software generates the same results on every compilation for the same source code and settings on a given system. Therefore, it is typically not necessary to take any steps to preserve your results from compilation to compilation. When changes are made to the source code or to the settings, you usually get the best results by allowing the software to compile without using any previous compilation results or location assignments. In some cases, if you avoid running **Analysis & Synthesis**, or **quartus_map**, and run the Fitter or another desired Quartus II executable instead, you can skip the synthesis stage of the compile.

You can use the incremental compilation feature to preserve synthesis results for a particular partition of your design by choosing a netlist type of post-synthesis.



You should use the incremental compilation flow to preserve compilation results instead of the LogicLock back-annotation flow described here.



For information about the incremental compilation design methodology, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

If you wish, you may preserve the nodes resulting from netlist optimizations. Preserving the nodes may be required if you use the LogicLock flow to back-annotate placement and/or import one design into another. (Note that this is not needed if you use the incremental compilation design flow along with the LogicLock feature).

If you are using any Quartus II synthesis netlist optimization options, you can save your optimized results. To do so, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings**. In the **Compilation Process Settings** page, turn on **Save a node-level netlist of the entire design into a persistent source file**. This option saves your final results as an atom-based netlist in Verilog Quartus Mapping file format. By default, the Quartus II software places the Verilog Quartus Mapping file in the **atom_netlists** directory under the current project directory. If you want to create a different Verilog Quartus Mapping file

using different Quartus II settings, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings**. In the **Compilation Process Settings** page, change the **File name** setting.

If you are using the synthesis netlist optimizations (and not any physical synthesis optimizations), generating a Verilog Quartus Mapping file is optional. To lock down the location of all logic and device resources in the design with or without a Quartus II-generated Verilog Quartus Mapping file, on the Assignments menu, click **Back-Annotate Assignments** and specify the desired options. You should use back-annotated location assignments unless the design has been finalized. Making any changes to the design invalidates your back-annotated location assignments. If you need to make changes later on, use the new source HDL code as your input files, and remove the back-annotated assignments corresponding to the old code or netlist.

If you create a Verilog Quartus Mapping file and wish to recompile the design, use the new Verilog Quartus Mapping file as the input source file and turn off the synthesis netlist optimizations for the new compilation.

Physical Synthesis Optimizations

Traditionally, the Quartus II design flow has involved separate steps of synthesis and fitting. The synthesis step optimizes the logical structure of a circuit for area, speed, or both. The Fitter then places and routes the logic cells to ensure critical portions of logic are close together and use the fastest possible routing resources. While this push-button flow produces excellent results, the synthesis stage is unable to anticipate the routing delays seen in the Fitter. Since routing delays are a significant part of the typical critical path delay, performing synthesis operations with physical delay knowledge allows the tool to target its timing-driven optimizations at these parts of the design. This tight integration of the fitting and synthesis processes is known as physical synthesis.

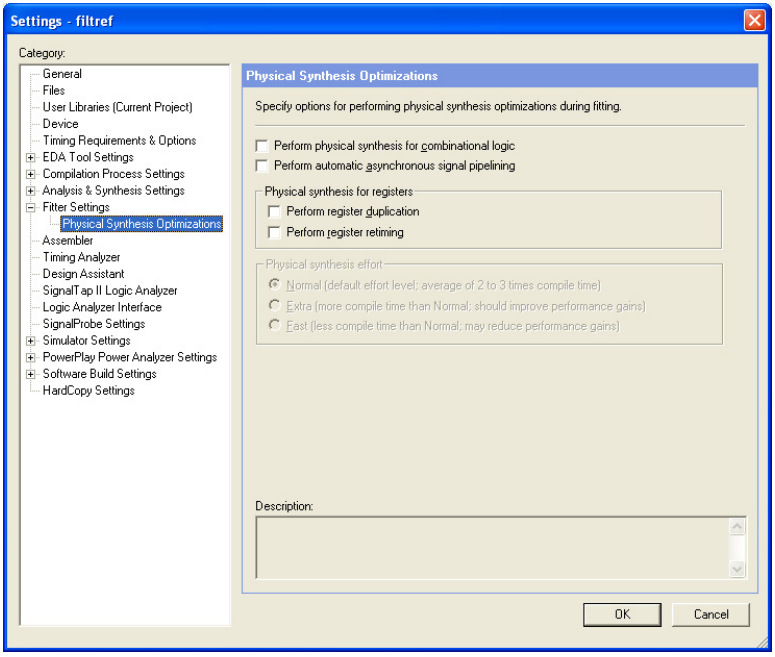
The following sections describe the physical synthesis optimizations available in the Quartus II software, and how they can help improve your performance results. Physical synthesis optimization options can be used with Arria GX, the Stratix and Cyclone series device families, as well as with HardCopy II devices.

If you are migrating your design to a HardCopy II device, you can target physical synthesis optimizations to the FPGA architecture in the FPGA-first flow or to the HardCopy II architecture in the HardCopy-first flow. The optimizations are mapped to the other device architecture during the migration process. Note that you cannot target optimizations to optimize for both device architectures individually because doing so would result in a different post-fitting netlist for each device.

For more information about using physical synthesis with HardCopy devices, refer to the *Quartus II Support of HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*.

To view and modify the physical synthesis optimization options, on the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings** and select **Physical Synthesis Optimizations**, as shown in Figure 11–7.

Figure 11–7. Physical Synthesis Optimization Settings



The physical synthesis optimizations are split into two groups: those that affect only combinational logic and not registers, and those that can affect registers. The options are split to allow you to keep your registers intact for formal verification or other reasons.

The following physical synthesis optimizations are available:

- Physical synthesis for combinational logic
- Automatic asynchronous signal pipelining
- Physical synthesis for registers:
  - Register duplication
  - Register retiming

You can control the effect of physical synthesis with the **Physical synthesis effort** option. The default selection is **Normal**. The **Extra** effort setting uses extra compilation time to try to achieve extra circuit performance, while the **Fast** effort setting uses less compilation time than **Normal** but may not achieve the same gains.

All Physical Synthesis optimizations write results to the **Netlist Optimizations** report. To access this report, on the Processing menu, click **Compilation Report**. In the **Category** list, select **Fitter** and select **Compilation Report**. This report provides a list of atom netlist files that were modified, created, and deleted during physical synthesis.

The node names for these atoms change during the physical synthesis process.

Nodes or entities that have the **Netlist Optimizations** logic option set to **Never Allow** are not affected by the physical synthesis algorithms. To access this logic option, on the Assignments menu, click **Assignment Editor**. Use this option to disable physical synthesis optimizations for parts of your design.

Automatic Asynchronous Signal Pipelining

The **Perform automatic asynchronous signal pipelining** option on the **Physical Synthesis Optimizations** page in the **Fitter Settings** section of the **Settings** dialog box allows the Quartus II Fitter to perform automatic insertion of pipeline stages for asynchronous clear and asynchronous load signals during fitting when these signals negatively affect performance. You can use this option if asynchronous control signal recovery and removal times are not achieving their requirements.

This option improves performance for designs in which asynchronous signals in very fast clock domains cannot be distributed across the chip fast enough due to long global network delays. This optimization performs automatic pipelining of these signals, while attempting to minimize the total number of registers inserted.



The **Perform automatic asynchronous signal pipelining** option adds registers to nets driving the asynchronous clear or asynchronous load ports of registers. This adds register delays (adds latency) to the reset, adding the same number of register delays for each destination using the reset, changing the behavior of the signal in the design. Therefore this option should only be used when adding latency to reset signals does not violate any design requirements. This option also prevents the promotion of signals to global routing resources.

The Quartus II software performs automatic asynchronous signal pipelining only if **Recovery/Removal Analysis** is enabled. Pipelining is allowed only on asynchronous signals that have the following properties:

- The asynchronous signal is synchronized to a clock (a synchronization register drives the signal)
- The asynchronous signal fans-out only to asynchronous control ports of registers

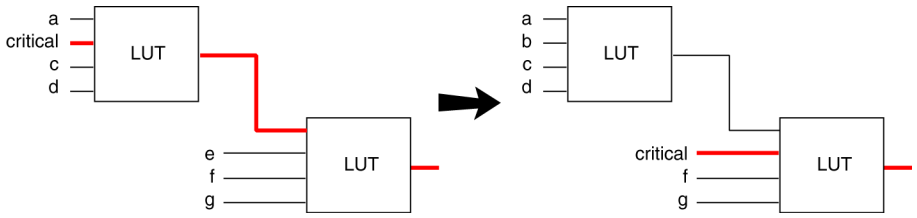
To access the **Recovery/Removal Analysis** option, on the Assignments menu, click **Settings**. In the **Category** list, select **Timing Requirements & Options**. On the **Timing Requirements & Options** page, click **More Settings**.

The Quartus II software does not perform automatic asynchronous signal pipelining on asynchronous signals that have the **Netlist Optimization logic** option set to **Never Allow**.

Physical Synthesis for Combinational Logic

To resynthesize the design and reduce delay along the critical path using the Quartus II Fitter, on the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings** and select **Physical Synthesis Optimizations**. In the **Physical Synthesis Optimizations** page, click **Perform physical synthesis for combinational logic**. The software can accomplish this type of optimization by swapping the look-up table (LUT) ports within LEs so that the critical path has fewer layers through which to travel. See Figure 11–8 for an example. This option also allows the duplication of LUTs to enable further optimizations on the critical path.

Figure 11–8. Physical Synthesis for Combinational Logic



In the first case, the critical input feeds through the first LUT to the second LUT. The Quartus II software swaps the critical input to the first LUT with an input feeding the second LUT. This reduces the number of LUTs contained in the critical path. The synthesis information for each LUT is altered to maintain design functionality.

The **Physical synthesis for combinational logic** option affects only combinational logic in the form of LUTs. The registers contained in the affected logic cells are not modified. Inputs into memory blocks, DSP blocks, and I/O elements (IOEs) are not swapped.

The Quartus II software does not perform combinational optimization on logic cells that have the following properties:

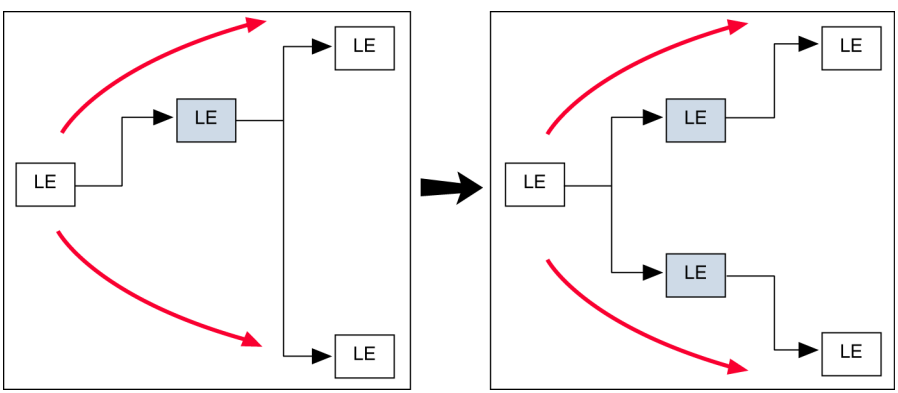
- Are part of a chain
- Drive global signals
- Are constrained to a single logic array block (LAB) location
- Have the **Netlist Optimizations** option set to **Never Allow**

If you want to consider logic cells with any of these conditions for physical synthesis, you can override these rules by setting the **Netlist Optimizations** logic option to **Always Allow** on a given set of nodes.

Physical Synthesis for Registers—Register Duplication

The **Perform register duplication** Fitter option on the **Physical synthesis Optimizations** page in the **Fitter Settings** section of the **Settings** dialog box allows the Quartus II Fitter to duplicate registers based on Fitter placement information. Combinational logic can also be duplicated when this option is enabled. A logic cell that fans out to multiple locations can be duplicated to reduce the delay of one path without degrading the delay of another. The new logic cell may be placed closer to critical logic without affecting the other fan-out paths of the original logic cell. Figure 11–9 shows an example of register duplication.

Figure 11–9. Register Duplication



The Quartus II software does not perform register duplication on logic cells that have the following properties:

- Are part of a chain
- Contain registers that drive asynchronous control signals on another register
- Contain registers that drive the clock of another register
- Contain registers that drive global signals
- Contain registers that are constrained to a single LAB location
- Contain registers that are driven by input pins without a t_{SU} constraint
- Contain registers that are driven by a register in another clock domain
- Are considered virtual I/O pins
- Have the **Netlist Optimizations** option set to **Never Allow**



For more information about virtual I/O pins, see the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*.

If you want to consider logic cells that meet any of these conditions for physical synthesis, you can override these rules by setting the **Netlist Optimizations** logic option to **Always Allow** on a given set of nodes.

Physical Synthesis for Registers—Register Retiming

The **Perform register retiming** fitter option in the **Physical Synthesis Optimizations** page in the **Fitter Settings** section of the **Settings** dialog box allows the Quartus II fitter to move registers across combinational logic to balance timing. This option enables algorithms similar to the **Perform gate-level register retiming** option (see "Gate-Level Register Retiming" on page 11–5). This option applies to the atom level (registers and combinational logic have already been placed into logic cells), and it compliments the synthesis gate-level option.

The Quartus II software does not perform register retiming on logic cells that have the following properties:

- Are part of a cascade chain
- Contain registers that drive asynchronous control signals on another register
- Contain registers that drive the clock of another register
- Contain registers that drive a register in another clock domain
- Contain registers that are driven by a register in another clock domain
- Contain registers that are constrained to a single LAB location
- Contain registers that are connected to SERDES
- Are considered virtual I/O pins
- Registers that have the **Netlist Optimizations** logic option set to **Never Allow**



For more information about virtual I/O pins, refer to the *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook.*

If you want to consider logic cells that meet any of these conditions for physical synthesis, you can override these rules by setting the **Netlist Optimizations** logic option to **Always Allow** on a given set of registers.

Preserving Your Physical Synthesis Results

Given the same source code and settings on a given system, the Quartus II software generates the same results for every compilation. Therefore, it is typically not necessary to take any steps to preserve your results from compilation to compilation. When changes are made to the source code or to the settings, you usually get the best results by allowing the software to compile without using any previous compilation results or location assignments. However, if you do wish to preserve the compilation results, make sure to follow the guidelines outlined in this section.

You can use the incremental compilation feature to preserve fitting results for a particular partition of your design by choosing a netlist type of post-fit.



You should use the incremental compilation flow to preserve compilation results instead of the LogicLock back-annotation flow described here.



For information about the incremental compilation design methodology, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

If you wish, you can preserve the nodes resulting from physical synthesis. Preserving the nodes may be required if you use the LogicLock flow to back-annotate placement and/or import one design into another. (Note that this is not needed if you use the incremental compilation design flow along with the LogicLock feature).

If you are using any Quartus II physical synthesis optimization options, you can save the nodes in your optimized result using the **Save a node-level netlist into a persistent source file (Verilog Quartus Mapping File)** option on the **Compilation Process Settings** page in the **Settings** dialog box. This option saves your final results as an atom-based netlist in Verilog Quartus Mapping file format. By default, the Quartus II software places the Verilog Quartus Mapping file in the **atom_netlists** directory under the current project directory. If you want to create a different Verilog Quartus Mapping file using different Quartus II settings, you may do so by changing the **File name** setting on the **Compilation Process Settings** page in the **Settings** dialog box.

If you are using the physical synthesis optimizations and you wish to lock down the location of all LEs and other device resources in the design using the **Back-Annotate Assignments** command, a Verilog Quartus Mapping file netlist is required to preserve the changes that were made to your original netlist. Since the physical synthesis optimizations depend on the placement of the nodes in the design, back-annotating the placement changes the results from physical synthesis. Changing the results means that node names are different, and your back-annotated locations are no longer valid. To access this option, on the Assignments menu, click **Back-Annotate Assignments**.

You should not use a Quartus II-generated Verilog Quartus Mapping file or back-annotated location assignments with physical synthesis optimizations unless the design has been finalized. Making any changes to the design invalidates your physical synthesis results and back-annotated location assignments. If you need to make changes later, use the new source HDL code as your input files, and remove the back-annotated assignments corresponding to the Quartus II-generated Verilog Quartus Mapping file.

To back-annotate logic locations for a design that was compiled with physical synthesis optimizations, first create a Verilog Quartus Mapping file. When recompiling the design with the hard logic location assignments, use the new Verilog Quartus Mapping file as the input source file and turn off the physical synthesis optimizations for the new compilation.

If you are importing a Verilog Quartus Mapping file and back-annotated locations into another project that has any **Netlist Optimizations** turned on, it is important to apply the **Netlist Optimizations** = **Never Allow** constraint, to make sure node names don't change, otherwise the back-annotated location or LogicLock assignments are invalid.



You should use the incremental compilation flow to preserve compilation results instead of using logic back-annotation.

Applying Netlist Optimization Options

Netlist optimizations options can have various effects on different designs. Designs that are well coded or have already been restructured to balance critical path delays may not see a noticeable difference in performance.

To obtain optimal results when using netlist optimization options, you may need to vary the options applied to find the best results. By default, all options are off. Turning on additional options leads to the largest effect on the node names in the design. Take this into consideration if you are using a LogicLock or verification flow such as the SignalTap II logic analyzer or formal verification that requires fixed or known node names. On average, applying all of the **physical synthesis** options at the **Extra** effort level produces the best results for those options, but adds significantly to the compilation time. You can also use the **Physical synthesis effort** option to decrease the compilation time.

The synthesis netlist optimizations typically do not add much compilation time, relative to the overall design compilation time.



When you are using a third-party atom netlist (Verilog Quartus Mapping file or Electronic Design Interchange Format file), the **WYSIWYG Primitive Resynthesis** option must be turned on in order to use the **Gate-level Register Retiming** option.

The Design Space Explorer (DSE) tool command language (Tcl)/Tk script is provided with the Quartus II software to automate the application of various sets of netlist optimization options.



For more information about using the DSE script to run multiple compilations, refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*. For information about typical performance results using combinations of netlist optimization options and other optimization techniques, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Scripting Reference Manual* includes the same information in PDF form.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

You can specify many of the options described in this section either on an instance, or at a global level, or both.

Use the following Tcl command to make a global assignment:

```
set_global_assignment -name <QSF variable name> <value>
```

Use the following Tcl command to make an instance assignment:

```
set_instance_assignment -name <QSF variable name> <value> -to <instance name>
```

Synthesis Netlist Optimizations

Table 11–1 lists the Quartus II Settings File (**.qsf**) variable name and applicable values for the settings discussed in "Synthesis Netlist Optimizations" on page 11–3. The Quartus II Settings File variable name

is used in the Tcl assignment to make the setting along with the appropriate value. The Type column indicates whether the setting is supported as a global setting, an instance setting, or both.

Table 11–1. Synthesis Netlist Optimizations and Associated Settings

Setting Name	Quartus II Settings File Variable Name	Values	Type
Perform WYSIWYG Primitive Resynthesis	ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP	ON, OFF	Global, Instance
Optimization Technique	<*Device Family Name*>_OPTIMIZATION_TECHNIQUE	AREA, SPEED, BALANCED	Global, Instance
Perform Gate-Level Register Retiming	ADV_NETLIST_OPT_SYNTH_GATE_RETIME	ON, OFF	Global
Power-Up Don't Care	ALLOW_POWER_UP_DONT_CARE	ON, OFF	Global
Allow Register Retiming to trade off Tsu/Tco with Fmax	ADV_NETLIST_OPT_RETIME_CORE_AND_IO	ON, OFF	Global
Save a node-level netlist into a persistent source file	LOGICLOCK_INCREMENTAL_COMPILE_ASSIGNMENT	ON, OFF	Global
	LOGICLOCK_INCREMENTAL_COMPILE_FILE	<*filename*>	
Allow Netlist Optimizations	ADV_NETLIST_OPT_ALLOWED	"ALWAYS ALLOW", DEFAULT, "NEVER ALLOW"	Instance

Physical Synthesis Optimizations

Table 11–2 lists the Quartus II Settings File variable name and applicable values for the settings discussed in "Physical Synthesis Optimizations" on page 11–11. The Quartus II Settings File variable name is used in the Tcl assignment to make the setting, along with the appropriate value. The Type column indicates whether the setting is supported as a global setting, an instance setting, or both.

Table 11–2. Physical Synthesis Optimizations and Associated Settings (Part 1 of 2)

Setting Name	Quartus II Settings File Variable Name	Values	Type
Physical Synthesis for Combinational Logic	PHYSICAL_SYNTHESIS_COMBO_LOGIC	ON, OFF	Global
Automatic Asynchronous Signal Pipelining	PHYSICAL_SYNTHESIS_ASYNCHRONOUS_SIGNAL_PIPELINING	ON, OFF	Global

Table 11–2. Physical Synthesis Optimizations and Associated Settings (Part 2 of 2)

Setting Name	Quartus II Settings File Variable Name	Values	Type
Perform Register Duplication	PHYSICAL_SYNTHESIS_REGISTER_DUPLICATION	ON, OFF	Global
Perform Register Retiming	PHYSICAL_SYNTHESIS_REGISTER_RETIMING	ON, OFF	Global
Power-Up Don't Care	ALLOW_POWER_UP_DONT_CARE	ON, OFF	Global, Instance
Power-Up Level	POWER_UP_LEVEL	HIGH, LOW	Instance
Allow Netlist Optimizations	ADV_NETLIST_OPT_ALLOWED	"ALWAYS ALLOW", DEFAULT, "NEVER ALLOW"	Instance
Save a node-level netlist into a persistent source file	LOGICLOCK_INCREMENTAL_COMPILE_ASSIGNMENT	ON, OFF	Global
	LOGICLOCK_INCREMENTAL_COMPILE_FILE	<*filename*>	

Incremental Compilation

For information about scripting and command line usage for incremental compilation as mentioned in "Preserving Synthesis Netlist Optimization Results" on page 11–10 or "Preserving Your Physical Synthesis Results" on page 11–17, refer to the *Quartus II Incremental Compilation* chapter in volume 1 of the *Quartus II Handbook*.

Back-Annotating Assignments

You can use the `logiclock_back_annotate` Tcl command to back-annotate resources in your design. This command can back-annotate resources in LogicLock regions, and resources in designs without LogicLock regions.

For more information about back-annotating assignments, see "Preserving Synthesis Netlist Optimization Results" on page 11–10 or "Preserving Your Physical Synthesis Results" on page 11–17.

The following Tcl command back-annotates all registers in your design.

```
logiclock_back_annotate -resource_filter "REGISTER"
```

The `logiclock_back_annotate` command is in the `backannotate` package.

Conclusion

Synthesis netlist optimizations and physical synthesis optimizations work in different ways to restructure and optimize your design netlist. Taking advantage of these Quartus II netlist optimizations can help improve your quality of results.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*
- *LogicLock Design Methodology* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Quartus II Support of HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

The following table shows the revision history for this chapter.

Document Revision History		
Date and Document Version	**Changes Made**	**Summary of Changes**
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 11–23.	—
May 2007 v7.1.0	• Added Arria GX as a newly supported device • Added Referenced Documents.	Updated chapter to include support for Arria GX.
March 2007 v7.0.0	No changes made.	—
November 2006 v6.1.0	Added revision history for document.	—
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0.	—
October 2005 v5.1.0	Chapter 11 was formerly Chapter 9 in version 5.0.	—

Document Revision History		
Date and Document Version	**Changes Made**	**Summary of Changes**
May 2005 v5.0.0	Chapter 9 was formerly Chapter 8 in version 4.2.	—
December 2004 v2.1	Updated for Quartus II software version 4.2: • General formatting and editing updates. • Additional description about fixed and primitive node names for synthesis netlist optimization and physical synthesis options. • Updates to figures. • Clarified APEX support. • Added information about node name changes for atoms during physical synthesis. • Deleted Physical Synthesis Report section.	—
June 2004 v2.0	• Updates to tables and figures. • New functionality in the Quartus II software version 4.1.	—
Feb. 2004 v1.0	Initial release.	—

12. Design Space Explorer

QII52008-7.2.0

Introduction

The Quartus® II software includes many advanced optimization algorithms to help you achieve timing closure, optimize area, and reduce dynamic power. The various settings and parameters control the behavior of the algorithms. These options provide complete control over the Quartus II software optimization and power techniques.

Each FPGA design is unique. There is no standard set of options that always results in the best performance or power utilization. Each design requires a unique set of options to achieve optimal performance. This chapter describes the Design Space Explorer (DSE), a utility written in Tcl/Tk that automates finding the best set of options for your design. DSE explores the design space of your design by applying various optimization techniques and analyzing the results.

DSE Concepts

This section explains the concepts and terminology used by DSE.

Exploration Space and Exploration Point

Before DSE explores a design, DSE creates an exploration space, which consists of Synthesis and Fitter settings available in the Quartus II software. Each group of settings in an exploration space is referred to as a *point*. An exploration space contains one or more points. DSE traverses the points in the exploration space to determine optimal settings for your design.

Seed and Seed Sweeping

The Quartus II Fitter uses a seed to specify the starting value that randomly determines the initial placement for the current design. The seed value can be any non-negative integer value. Changing the starting value may or may not produce better fitting. However, varying the value of the seed or seed sweeping allows the Quartus II software to determine an optimal value for the current design.

DSE extends Fitter seed sweeping in exploration spaces by providing a method for sweeping through general compilation and Fitter parameters to find the best options for your design. You can run DSE in various exploration space modes, ranging from an exhaustive try-all-options-and-values mode to a mode that focuses on one parameter.

DSE Exploration

DSE compares all exploration point results with the results of a base compilation, generated from the initial settings that you specify in the original Quartus II project files. As DSE traverses all points in the exploration space, all settings, not explicitly modified by DSE, default to the base compilation setting. For example, if an exploration point turns on register retiming but does not modify the register packing setting, the register packing setting defaults to the value you specified in the base compilation.



DSE performs the base compilation with the settings you specified in the original Quartus II project. These settings are restored after DSE traverses all points in the exploration space.

General Description

You can use DSE in either the graphical user interface (GUI) or from a command line. To run DSE with the GUI, either click **Design Space Explorer** on the Tools menu in the Quartus II software, or at the command prompt, type:

```
quartus_sh --dse ↵
```

To run DSE from a command line, type the following command at the command prompt:

```
quartus_sh --dse -nogui [<options>] ↵
```

You can run DSE with the following options:

```
-archive
-concurrent-compiles [0..6]
-custom-file <filename>
-decision-column <"column name">
-exploration-space <"space">
-ignore-failed-base
-ignore-signalprobe
-ignore-signaltap
-llr-restructuring
-lower-priority
-lsf-queue <queue name>
-nogui
-optimization-goal <"goal">
-project <project name>
-revision <revision name>
-run-power
-search-method <"method">
-seeds <seed list>
-skip-base
-slaves <"slave list">
-stop-after-time <dd:hh:mm>
-stop-after-zero-failing-paths
-use-lsf
```

The DSE script is in the default Quartus II software installation in <*Quartus II installation directory*>**/common/tcl/apps/dse/dse.tcl** on the PC, Solaris, HP-UX, and Linux platforms. You can launch DSE using one of the following methods:

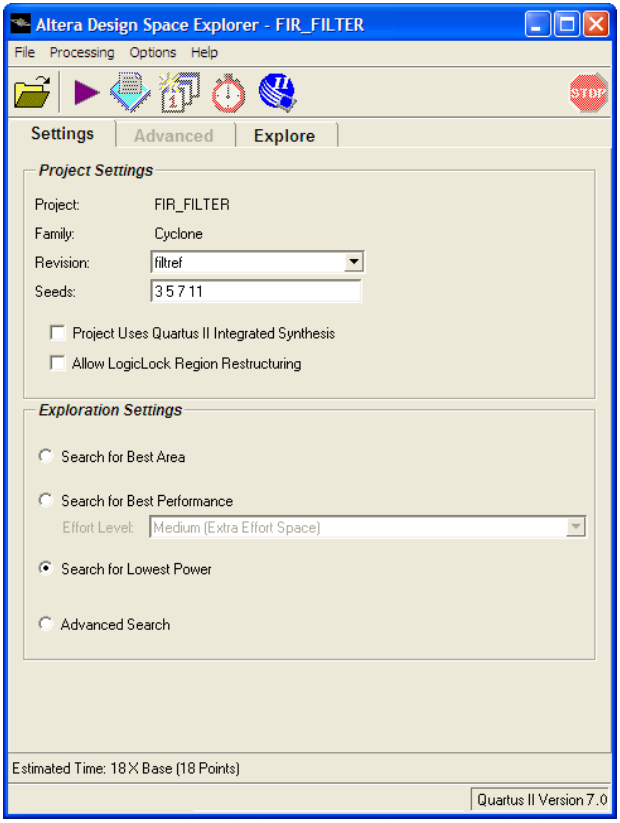
- On the Tools menu, click **Launch Design Space Explorer**.
- On Windows, select **Start** > **Programs** > **Altera** > **Design Space Explorer** or **Quartus II** <*version number*>.



For more information on DSE, launch the DSE GUI. On the Help menu, click **Contents** or press the **F1** key.

Figure 12–1 shows the DSE user interface. The **Settings** tab is divided into two sections: **Project Settings** and **Exploration Settings**.

Figure 12–1. DSE User Interface



Timing Analyzer Support

DSE supports both the Quartus II Classic Timing Analyzer and the Quartus II TimeQuest Timing Analyzer. You must set the timing analyzer prior to opening the project in DSE. Once the timing analyzer is set, DSE performs the design exploration with the selected timing analyzer that guides the fitter.



TimeQuest is launched directly from DSE if you set the default timing analyzer to TimeQuest.

DSE Flow

You can run DSE at any point in the design process. However, Altera recommends that you run DSE late in your design cycle when you are focusing on optimizing performance and power. The results gained from different combinations of optimization options may not persist over large changes in a design. Running DSE in signature mode (refer to "Signature Mode" on page 12–13) at the midpoint of your design cycle shows you the affect of various parameters such as the register packing logic option on your design.

DSE runs the Quartus II software for every compilation specified in the **Exploration Settings** options. DSE selectively determines the best settings for your design based on the **Optimization Goal** selected for the exploration. The Quartus II software always attempts to achieve all your timing requirements regardless of the Optimization Goal set in DSE. The Optimization Goal changes the metrics that DSE evaluates to determine if one compilation is better than another. Design Space Explorer does not change the behavior of the Quartus II software.

DSE reports the compilation that has the smallest slack. Specifying all timing requirements before you use DSE to explore your design is very important to ensure that DSE finds the optimal set of parameters for your design based on design criteria you set in your initial design.

You can change the initial placement configuration used by the Quartus II Fitter by varying the **Fitter Seed** value. You can enter seed values in the **Seeds** field of the DSE user interface.



To set the seed value on the Assignments menu, click **Fitter Settings** in the **Settings** dialog box.

Compilation time increases as DSE exploration spaces become more comprehensive. Increased compilation time results from running several compilations and comparing the generated results with the original base compilation results.

For typical designs, varying only the seed value results in a 5% f_{MAX} increase. For example, when compiling with three different seeds, one-third of the time f_{MAX} does not improve over the initial compilation, one-third of the time f_{MAX} gets 5% better, and one-third of the time f_{MAX} gets 10% better.

DSE Support for Altera Device Families

DSE setting support varies across device families. To see the range of settings DES supports, click the **Advanced Search** radio button on the **Settings** tab, then select the **Advanced** tab to access the settings listed in the following categories:

- Exploration Space
- Optimization Goal
- Search Method

The following device families support all **Advanced** setting types:

- Arria™ GX
- Stratix® III
- Stratix II
- Stratix
- Stratix GX
- Cyclone® III
- Cyclone II
- Cyclone
- MAX® II

The following device families support only the **Advanced Exploration Space** and **Optimization Goal** settings shown in Table 12–1:

- APEX™ 20K
- APEX 20KC
- APEX 20KE
- APEX II
- FLEX 10K®
- FLEX® 10KA
- FLEX 10KE

Click the **Advanced Search** radio button on the **Settings** tab before you select the **Advanced** tab to access the settings in Table 12–1.

Table 12–1. Advanced Exploration Space Support for APEX 20K, APEX II and FLEX 10K Devices

Seed sweep	Area optimization space
Signature fitting effort level	Extra effort space
Extra effort for Quartus II Integrated Synthesis Projects	Custom space

DSE Project Settings

This section provides the following information about DSE project settings:

- Setting up the DSE work environment
- Specifying the revision
- Setting the initial seed
- Quartus II integrated synthesis
- Restructuring LogicLock regions

Setting Up the DSE Work Environment

From the DSE GUI, you can open a Quartus II project for a design exploration with either of the following actions:

- On the File menu, click **Open Project** and browse to your project.
- Use the **Open** icon to open a project.

Specifying the Revision

You can specify the revision to be explored with the **Revision** field in the DSE user interface. The **Revision** field is populated after the Quartus II project has been opened.



If no revisions were created in the Quartus II project, the default revision, which is the top-level entity, is used. For more information, refer to *Quartus II Project Management* chapter in volume 2 of the *Quartus II Handbook*.

Setting the Initial Seed

To specify the seed that DSE uses for an exploration, specify a non-negative integer value in the **Seed** box under **Project Settings** on the **Settings** tab. The seed value determines your design's initial placement in a Quartus II compilation.

To specify a range of seeds, type the low end of the range followed by a hyphen, followed by the high end of the range. For example, 2-5 specifies that DSE uses the values 2, 3, 4, and 5 as seeds.

Restructuring LogicLock Regions

The **Allow LogicLock Region Restructuring** option allows DSE to modify LogicLock region properties in your design, if any exist. DSE applies the **Soft** property to LogicLock regions to improve timing. In addition, DSE can remove LogicLock regions that negatively affect the performance of the design.



"Exploration Space" on page 12–10 describes the type of explorations you can perform.

Search for Best Performance, Search for Best Area Options, or Search for Lowest Power Option

Use the **Exploration Settings** list to select the type of exploration to perform: **Search for Best Area**, **Search for Best Performance**, **Search for Lowest Power**, or **Advanced Search**.

The **Search for Best Performance** option uses a predefined exploration space that targets performance improvements for your design. Depending on the device your design targets, you can select up to four predefined exploration spaces: **Low** (**Seed Sweep**), **Medium** (**Extra Effort Space**), **High (Physical Synthesis Space)**, and **Highest** (**Physical Synthesis with Retiming Space**). As you move from **Low** to **Highest**, the number of options explored by DSE increases, causing compilation time to increase.

The **Search for Lowest Power** option uses a predefined exploration space that targets overall power improvements for your design. When **Search for Lowest Power** is selected, DSE automatically runs the PowerPlay Power Analyzer for each point in the space. You must ensure that the PowerPlay Power Analyzer is configured correctly to ensure accurate results. DSE issues a warning if the confidence level for any power estimate is low.

The **Search for Best Area** option uses a predefined exploration space that targets device utilization improvements for your design.

Advanced Search Option

The **Advanced Search** option provides full control over the exploration space, the optimization goal for your design, and the search method used in a design exploration. Refer to "Performing an Advanced Search in Design Space Explorer" on page 12–9 for detailed information on how to set up and perform an **Advanced Search** in DSE.



You can use **Advanced Search** to define exploration spaces that are equivalent to the **Search for Best Area**, **Search for Lowest Power**, and **Search for Best Performance** options.

Quartus II Integrated Synthesis

The **Project Uses Quartus II Integrated Synthesis** option works only for designs that have been synthesized with Quartus II integrated synthesis. With this option turned on, DSE explores options that affect the synthesis stage of compilation.

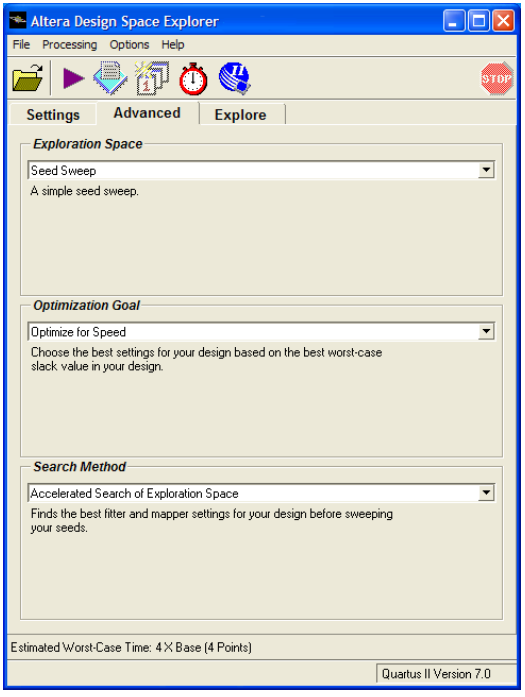
For more information on integrated synthesis options, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Performing an Advanced Search in Design Space Explorer

You must make three exploration settings in the **Advanced Search** dialog box before exploring a design. These three settings, **Exploration Space**, **Optimization Goal**, and **Search Method**, are described in the following sections. Figure 12–2 shows the **Advanced Search** dialog box.

You can access the **Advanced** tab only after you open a Quartus II project in DSE and select **Advanced Search** on the **Settings** tab.

Figure 12–2. DSE Advanced Search Dialog Box



Exploration Space

The **Exploration Space** list controls the types of explorations that DSE performs on your design. DSE traverses the points in the exploration space, applying the settings to the design and comparing compilation results to determine the best settings for your design. DSE offers the following exploration space types:

- Seed Sweep
- Extra Effort Spaces
- Physical Synthesis Spaces
- Retiming Spaces
- Area Optimization Space
- Selective Performance Optimization Space
- Custom Space
- Signature mode—Power Optimization Spaces



Not all **Advanced** exploration space types are available for every device family. Refer to "DSE Support for Altera Device Families" on page 12–6 for **Advanced** exploration space support for various device families.

Compilation time increases proportionally to the breadth of the explorations. The exploration space compilation time increases with the number and type of exploration spaces DSE explores, especially with exploration space types that have more optimization options and parameters.

On the Options menu, point to **Advanced**, and turn on **Save Exploration Space to File** to save an XML file representing the exploration space. DSE writes the exploration space to a file named **<***project name***>.dse** in the project directory. You can modify this file to create a custom exploration space.

For more information on using custom exploration spaces in DSE, refer to "Creating Custom Spaces for DSE" on page 12–22.

Seed Sweep

Enter the seed values in the **Seeds** field in the DSE user interface. There are no "magic" seeds. The variation between seeds is truly random, any non-negative integer value is as likely to produce good results. DSE defaults to seeds 3, 5, 7, and 11. The **Seed Sweep** exploration space does not change your netlist.



The **Seeds** field accepts individual seed values, for example, 2, 3, 4, and 5, or seed ranges, for example, 2-5.

Compilation time increases 1× for every seed value you specify. For example, if you enter five seeds, the compilation time increases to 5× the initial compilation time.

Extra Effort Spaces

The **Extra Effort Space** exploration space adds the **Register Packing** option to the exploration space done by the **Seed Sweep**. The **Extra Effort Space** exploration space also increases the Quartus II Fitter effort during placement and routing. However, the **Extra Effort Space** exploration space does not change your netlist.

Physical Synthesis Spaces

The **Physical Synthesis Space** exploration space adds physical synthesis options such as register retiming and physical synthesis for combinational logic to the options included in the **Extra Effort Space** exploration space. These netlist optimizations move registers in your design. Look-up tables (LUTs) are modified by these options. However, the design behavior is not affected by these options.



For more information about physical synthesis, refer to the *Netlist Optimizations & Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

The **Physical Synthesis for Quartus II Integrated Synthesis Projects** exploration space includes all the options in the **Physical Synthesis** exploration space and explores various Quartus II integrated synthesis optimization options. The **Physical Synthesis for Quartus II Integrated Synthesis Projects** exploration space works only for designs that have been synthesized using Quartus II integrated synthesis software.

Retiming Space

The **Physical Synthesis with Retiming Space** exploration space includes all the options in the **Physical Synthesis Space** exploration space and explores register retiming. Register retiming can move registers in your design.

The **Physical Synthesis Retiming Space for Quartus II Integrated Synthesis Projects** exploration space includes all the options in **Physical Synthesis with Retiming Space** exploration space, and also explores various Quartus II integrated synthesis optimization options. The **Physical Synthesis with Retiming Space for Quartus II Integrated Synthesis Projects** exploration space works only for designs that have been synthesized using the Quartus II integrated synthesis.

Area Optimization Space

The **Area Optimization Space** exploration space explores options that affect logic cell utilization for your design. These options include register packing and **Optimization Technique** set to **Area**.

Selective Performance Optimization Space

The Selective Performance Optimization combines a seed sweep with various performance fitter settings to improve the timing of your design. The seed sweep is performed over a limited number of points in such a way that the base settings are not replicated. This is the recommended option for large designs where other spaces may be too large.

Custom Space

Use the **Custom Space** exploration space to selectively explore the effects of various optimization options on your design. This exploration space gives you complete control over which options are explored and in what mode. In the **Custom Space** mode you can explore all optimization options available in DSE.

Table 12–2 shows the settings adjusted by each exploration space.

Table 12–2. Summaries of Exploration Spaces *Note (1)*

Search Type	Exploration Spaces					
	Seed Sweep	Extra Effort	Physical Synthesis	Retiming	Area Optimization	Custom
Analysis and Synthesis Settings						
Optimization technique	—	—	✓	✓	✓	✓
Perform WYSIWYG resynthesis	—	—	✓	✓	✓	✓
Perform gate-level register retiming	—	—	—	✓	—	✓
Fitter Settings						
Fitter seed	✓	✓	✓	✓	✓	✓
Register packing	—	✓	✓	✓	✓	✓
Increase PowerFit fitter effort	—	✓	✓	✓	—	✓
Perform physical synthesis for combinational logic	—	—	✓	✓	—	✓
Perform register retiming	—	—	—	✓	—	✓

Note to Table 12–2:
(1) For exploration spaces that includes Quartus II Integrated Synthesis Projects, DSE increases the synthesis effort.

For more information about using custom exploration spaces with DSE, refer to "Creating Custom Spaces for DSE" on page 12–22.

Signature Mode

In **Signature** mode, DSE analyzes the f_{MAX}, slack, compilation time, and area trade-offs of a single parameter. Running the single parameter over multiple seeds, DSE reports the average of the resulting values. With this information you gain a better understanding of how that parameter affects your design. There are four signature mode settings in DSE:

- Signature: Fitting Effort Level
- Signature: Netlist Optimizations
- Signature: Fast Fit
- Signature: Register Packing

Each setting explores a specific optimization option for your design. For example, in **Signature: Register Packing** mode, DSE explores the **Auto Packed Registers** logic option with its four settings (**OFF, Normal**, **Minimized Area**, and **Minimize Area with Chains**), and reports the effects of each on your design.

Optimization Goal

Design metrics are extremely important in exploring your design, whether the metric is performance, logic utilization, or a combination of both. These metrics allow you to determine which compilation is best, based on the design requirements. By specifying options in the **Optimization Goal** settings, you specify your optimization design goals. DSE then uses the **Optimization Goal** settings to determine the best compilation results. Table 12–3 summarizes the six available optimization settings.

Table 12–3. Optimization Goal Settings (Part 1 of 2)

Setting	Description
Optimize for Speed	The exploration point containing the smallest worst-case slack value is selected as the best run.
Optimize for Area	The exploration point containing the lowest logic cell count is selected as the best run
Optimize for Power	The exploration point containing the lowest thermal power dissipation, and, if possible, a positive worst-case slack value, is selected as the best run.
Optimize for Negative Slack and Failing Paths	The exploration point containing the best average negative worst-case slack and lowest number of failing paths is selected as the best run.

Table 12–3. Optimization Goal Settings (Part 2 of 2)	
Setting	**Description**
Optimize for Average Period	The exploration point containing the highest average period in a multiclock design is selected as the best run.
Optimize for Quality Fit	The exploration point containing the highest quality of fit is selected as the best run.

Quality of Fit (QoF)

Quality of Fit (QoF) is a better evaluation of fit than traditional worst-case slack metrics, because QoF considers all timing domains. QoF is not susceptible to the common mistake of accepting a fit because it has marginally better worst-case slack than other marginal timing domains with much worse slack. For example, the traditional worst-case slack metric favors a fit that achieves -2 ns slack for clock A and -5 ns slack for clock B, over a fit that achieves 1 ns slack for clock A and -5.5 ns slack for clock B. By applying a piece-wise linear function to each domain slack value, QoF ensures that large improvements in domains with ample slack do not unnecessarily skew the overall quality assessment of the fit.

To achieve a representative QoF value, ensure that slack values from domains that are easily meeting timing requirements do not offset the slack values from domains that are marginally meeting timing requirements. To correlate these values correctly, DSE applies a piece-wise linear function to the individual slack values before they are added together. This function reduces the improvement per unit of additional slack in a domain, as the domain slack improves. For example, the improvement of 100 ps in a domain that begins with 0 ns of slack is weighted more significantly than a 100 ps improvement in a domain that begins with 10 ns of slack.

To calculate the QoF for a design, use the sum of worst- case slack values for all timing domains reported by timing analysis. Timing domains include: Clock Setup, Clock Hold, t_{SU}, t_{CO}, t_{PD}, t_{H}, min t_{CO}, min t_{PD}, and other timing parameters. For example, if clock A has a Clock Setup slack of -500 ps, and clock B has a Clock Setup slack of 200 ps, the QoF for these two domains is -700 ps. The higher the QoF value reported, the better the QoF.

The QoF can be calculated for any fully compiled design by entering the following Tcl command in the Tcl console:

```
source [file join $::quartus(tclpath) apps dse
calculate_quality_of_fit.tcl]
```



All variables in the above statement are predefined; type the statement as shown without any variable substitution.

Search Method

The **Search Method** setting allows you to control the breadth of the search that DSE performs. DSE provides two search methods: **Exhaustive search of exploration space** and **Accelerated search of exploration space**. These search methods are described in Table 12–4.

Table 12–4. Search Methods

Search Method	Description
Exhaustive search of exploration space	Applies all settings available in the exploration space to all seeds specified. This search method yields optimal settings for your design, but this search requires the most time.
Accelerated search of exploration space	Finds the best exploration space for your design by first determining the best settings and then sweeping the settings across all seeds specified.

DSE Flow Options

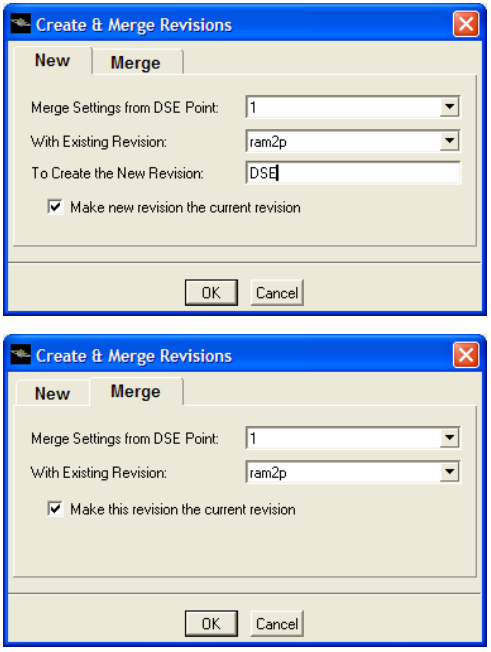
You can control the configuration of DSE with the following options:

- Create a Revision from a DSE Point
- Change Decision Column
- Stop Flow When Zero Failing Paths are Achieved
- Continue Exploration Even If Base Compilation Fails
- Run Quartus II PowerPlay Power Analyzer During Exploration
- Archive All Compilations
- Stop Flow After Time
- Save Exploration Space to File
- Ignore SignalTap and SignalProbe Settings
- Skip Base Analysis and Compilation If Possible
- Lower Priority of Compilation Threads
- DSE Configuration File

Create a Revision from a DSE Point

After you have performed a design exploration with DSE, a Quartus II revision can be made from any exploration point. This option facilitates the creation of multiple revisions based on the same space point for further optimization within the Quartus II software. Figure 12–3 shows the **Create a Revision From a DSE Point** dialog box.

Figure 12–3. Create a Revision from a DSE Point



Change Decision Column

The criteria DSE uses to determine the best space point in an exploration is known as the **Decision** column. As DSE explores a design space, the best exploration point changes according to the following inequality:

<Current Decision Column Value> > <Previous Decision Column Value>

By default, DSE uses worst-case slack as the **Decision** column for an exploration. The worst-case slack **Decision** column is the greatest slack value in an exploration, which can be I/O timing or clock slack values. You can change the **Decision** column on the Options menu. On the

Options menu, click **Advanced**, and select **Change Decision Column**. Table 12–5 lists the available **Decision** columns. The **Decision** column can be any column within the Quartus II Timing Analyzer Report.

Table 12–5. DSE Change Decision Columns

Decision Column Name	Description
Worst-case slack (default)	Determines best exploration point on worst-case slack in the exploration space.
Clock Setup: '<*clock name*>': Slack	Determines best exploration point on the <*clock name*> specified.
Clock Setup: '*': Slack	Determines best exploration point on all clocks.
Worst-case minimum t_{CO} Slack	Determines best exploration point on worst case minimum t_{CO} slack.
Worst-case t_H Slack	Determines best exploration point on worst-case t_H slack.
Worst-case t_{SU} Slack	Determines best exploration point on worst case t_{SU} slack.
<*any column name*>	Determines best exploration point on any column available in the Quartus II timing analysis report file.

Stop Flow When Zero Failing Paths are Achieved

Instructs DSE to stop exploring the space after it encounters any point, including the base point, that has zero failing paths. DSE uses the failing path count reported in the **All Failing Paths report** column to make this decision.

Continue Exploration Even If Base Compilation Fails

With the **Continue Exploration Even If Base Compilation Fails** option turned on, DSE continues the exploration even when a design compilation error occurs. For example, if timing settings are not applied to your design, a DSE error occurs. To cause DSE to continue with the exploration instead of halting when an error occurs, turn on this option.

Run Quartus II PowerPlay Power Analyzer During Exploration

Turn on **Run Quartus II PowerPlay Power Analyzer During Exploration** to invoke the Quartus II PowerPlay Analyzer for every exploration performed by DSE. Using this option can help you debug your design and determine trade-offs between power requirements and performance optimization.

Archive All Compilations

Turn on **Archive All Compilations** to create a Quartus II Archive File (.**qar**) for each compilation. These archive files are saved to the **dse** directory in the design's working directory.

Stop Flow After Time

Turn on **Stop Flow After Time** to stop further exploration after a specified number of days, hours, and/or minutes.



Exploration time might exceed the specified value because DSE does not stop in the middle of a compilation.

Save Exploration Space to File

Turn on **Save Exploration Space to File** to write out a <*project name*>**.dse** file containing all options explored by DSE. You can use or modify this file to perform a custom exploration.

Ignore SignalTap and SignalProbe Settings

DSE uses advanced physical synthesis options that are not compatible with the SignalTap® II or SignalProbe™ features. As a result, DSE issues an error message when a project is opened for exploration that has either SignalTap II or SignalProbe turned on. The error message is similar to the following:

```
Error Opening Project--------------------------
Project is using SignalProbe. Please turn off
SignalProbe before using this project with Design Space
Explorer or Ignore SignalProbe Setting in your Design
on the Options menu.
```

When the **Ignore SignalTap and SignalProbe Settings** option is turned on, DSE bypasses this check.

If you have already verified the design, you might save compilation time and improve resource utilization by turning this option on.

Skip Base Analysis and Compilation If Possible

Skip Base Analysis and Compilation If Possible allows the DSE to skip the Analysis and Elaboration stage or the compilation of the base point if base point compilation results are available from a previous Quartus II compilation.

Lower Priority of Compilation Threads

The **Lower Priority of Compilation Threads** option allows DSE to run the Quartus II executables with the `lower_priority` option. The `lower_priority` option lowers the priority of the Quartus II executable.

DSE Configuration File

Many options exist that allow you to customize the behavior of each DSE exploration. For example, you can specify seed values or a list of slave computers to be used for a distributed exploration run. Each time you close the DSE GUI it saves these values in a configuration file, **dse.conf**. The net time you launch the DSE GUI, it reads the values from **dse.conf** and restores the previous exploration settings.

Where the **dse.conf** file is stored varies based on the operating system that launches DSE. Table 12–6 specifies the locations where **dse.conf** files are stored based on operating system usage.

Table 12–6. DSE Configuration File Location

OS	File Location (default)	Comment
Windows	%APPDATA%/Altera/dse.conf	If the variable %APPDATA% is not defined, the configuration file is saved to **/.altera.quartus/dse.conf**
Unix	**~/.altera.quartus/dse.conf**	



Settings specified in the DSE command-line mode are not saved to a **dse.conf** configuration file.

DSE Advanced Information

This section covers advanced features that are available in DSE. These features increase the processing efficiency of design space exploration and provide further customization of the design space.

Computer Load Sharing in DSE Using Distributed Exploration

When you select **Distribute Compiles to Other Machines**, the DSE uses cluster computing technology to decrease exploration time. DSE uses multiple client computers to compile points in the specified exploration space. When you select the **Distributed DSE** option, DSE functions in one of the following operation modes:

- **Use LSF Resources**: DSE uses the Platform LSF grid computing technology to distribute exploration space points to a computing network.
- **Distribute Compiles to Other Machines** uses a Quartus II master process: DSE acts as a master and distributes exploration space points to client computers.

Distributed DSE Using LSF Resources

The easiest way to use distributed DSE technology is to submit the compilations to a preconfigured LSF cluster at your local site. For more information on LSF software, refer to **www.platform.com**, or contact your system administrator. To run Distributed DSE using LSF resources, from the Options menu, select **Distributed DSE** and click the **Configure Resources** option.

Distributed DSE Using a Quartus II Master Process

Before DSE can use computers in the local area network to compile points in the exploration space, you must create Quartus II software slave instances on the computers that will be used as clients. Type the following command at a command prompt on a client computer:

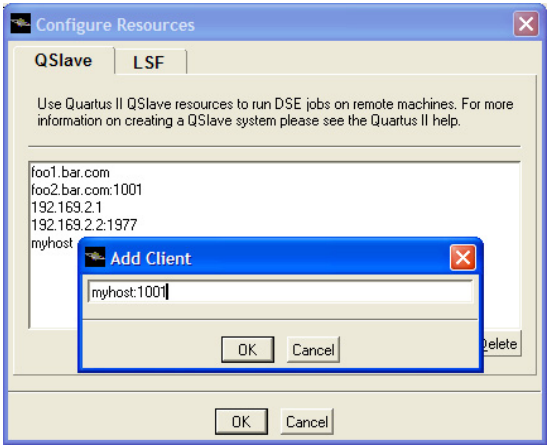
```
quartus_sh --qslave ↵
```

Repeating this on several computers creates a cluster of Quartus II software slaves for DSE to use. After you have created a set of Quartus II software slaves on the network, add the names of each slave computer in the QSlave tab of the **Configure Resources** dialog box.

To access the **Configure Resources** dialog box, from the Options menu, point to **Distributed DSE** and click **Configure Resources**. To add resources, click **Add** and type the client name. Click **OK**.

Figure 12–4 shows an example of client entries for a distributed search.

Figure 12–4. Client Entry in DSE



At the start of an exploration, DSE assumes the role of a Quartus II software master process and submits points to the slaves on the list to compile. If the list is empty, DSE issues an error and the search stops.

For more information about running and configuring Quartus slaves, at the command prompt, type:

```
quartus_sh --help=qslave ↵
```

Distributed DSE uses a protocol based on FTP to move files between the master and the slaves. By default, the qslave client listens to port number 1977 for communication with the master. If you are running a firewall on a machine that is running the qslave client, make sure you configure the firewall software such that it allows incoming and outgoing transmission control protocol (TCP) and user datagram protocol (UDP) packets on the port used by qslave.

You must set this up in every machine that is used as a slave in a distributed DSE environment.

You can change the default port number used by qslave by running the following command:

```
quartus_sh --qslave port=<new_port_number> ↵
```

You must use the same version of the Quartus II software to run the slave processes as you use to run DSE. To determine which Quartus II software version that you are using to run DSE, select Help and click **About DSE.** Unexpected results can occur if you mix different Quartus II software versions when using the Distributed DSE feature.

Concurrent Local Compilations

To reduce compilation time, DSE can compile exploration points concurrently. The **Concurrent Local Compilations** option allows you to specify the number of local compilations that DSE performs. For the **Concurrent Local Compilations** option, you can specify up to six concurrent compilations by choosing an integer value ranging from 1 through 6. You can use this option in conjunction with Distributed DSE. However, your system must have both the appropriate resources and licenses to perform concurrent compilations, and distributed processing. Multiprocessor or multicore systems are recommended for concurrent local compilations.



Concurrent Local Compilations require a separate Quartus II software license for each concurrent compilation. For example, if you compile four concurrent compilations, you need four licenses. Be sure before you choose a **Concurrent Local Compilations** value and start compilation that sufficient licenses are available.

Creating Custom Spaces for DSE

You can use custom spaces to explore combinations of options that are not in the predefined **Exploration Space** list. An exploration space is defined in an XML file. The following sections describe the tags you use to create a **Custom Space** that DSE can process.

A custom space is defined by the following three pairs of tags:

- `<DESIGNSPACE>` and `</DESIGNSPACE>`
- `<POINT>` and `</POINT>`
- `<PARAM>` and `</PARAM>`

DESIGNSPACE Tag

The `<DESIGNSPACE>` tag defines the start of the exploration space of a custom space. The end tag `</DESIGNSPACE>` defines the end of the exploration space. Both of these tags are required for all custom spaces.

POINT Tag

The POINT tag pair must occur within the DESIGNSPACE tag pair. The <POINT <*name*>="<*stage*>" enabled="<*value*>"> tag defines the start of the exploration point in a custom exploration space. The end tag </POINT> defines the end of the exploration point. The POINT tag also allows you to specify the <*stage*> value and whether a particular point is active for a particular DSE exploration.

The "<*stage*>" value in the POINT tag can be one of the following:

- **map**—indicates an Analysis and Synthesis setting change for that point
- **fit**—indicates a Fitter setting change for that point
- **seed**—indicates a Fitter seed change
- **llr**—indicates a LogicLock property change

The "<*value*>" value in the POINT tag can either be "**1**," indicating that for a specific stage the exploration point is active, or "**0**" for an inactive point.

An example of a POINT tag is shown in Example 12–1:

Example 12–1. Example of the POINT Tag

```
<POINT space="map" enabled="1">
...
</POINT>
```

The preceding point indicates a point that has Analysis and Synthesis setting changes and is active during Analysis and Synthesis.

PARAM Tag

The PARAM tag pair must occur within the POINT tag pair. The <PARAM name="<*parameter*>"> tag defines the start of a parameter to be modified for a particular exploration point. The end tag </PARAM> defines the end of the parameter.

An example of a PARAM tag is shown in Example 12–2:

Example 12–2. Example of the PARAM tag

```
<PARAM name +"ADV_NETLIST_OPT_SYNTH_GATE_RETIME"> ON </PARAM>
```

The **Analysis and Synthesis** settings and the "<*parameter*>" values are shown in Table 12–7.

Table 12–7. Analysis and Synthesis Settings *Note (1)*

Analysis and Synthesis Settings	Description	Value
<*device family name*>_OPTIMIZATION_TECHNIQUE	Type of optimization technique to use during the Quartus II Analysis and Synthesis stage for the specific device family.	SPEED, AREA, BALANCED
ADV_NETLIST_OPT_SYNTH_GATE_RETIME	Gate-level register retiming.	OFF, ON
ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP	WYSIWYG primitive resynthesis.	OFF, ON
DSE_SYNTH_EXTRA_EFFORT_MODE	Controls the Quartus II software synthesis effort.	MODE_1, MODE_2, MODE_3

Note to Table 12–7:
(1) Not all Analysis and Synthesis settings are available for all device families.

The point in Example 12–2 indicates that the Analysis and Synthesis setting gate-level retiming is turned on for the exploration space point.

Table 12–8 shows the Fitter settings.

Table 12–8. Fitter Settings *Note (1)*

Fitter Settings	Description	Value
AUTO_PACKED_REGISTERS_<*device family name*>	Register packing for the specified device family	NORMAL, MINIMIZE_AREA, MINIMIZE_AREA_WITH_CHAINS, AUTO, SPARSE, SPARSE_AUTO, OFF
INNER_NUM	PowerFit Fitter effort level	{integer value}
PHYSICAL_SYNTHESIS_COMBO_LOGIC	Physical synthesis for combinational logic	OFF, ON
PHYSICAL_SYNTHESIS_REGISTER_DUPLICATION	Physical synthesis for register duplication	OFF, ON
PHYSICAL_SYNTHESIS_REGISTER_RETIMING	Physical synthesis for register retiming	OFF, ON

Note to Table 12–8:
(1) Not all Fitter settings are available for all device families.

Simple Custom Space

Example 12–3 shows a simple custom exploration space performing a seed sweep with settings for the Analysis and Synthesis and the Fitter compilation stages.

Example 12–3. Simple Custom Exploration Space

```
<DESIGNSPACE>
    <POINT space="map" enabled="1">
    <PARAM name="CYCLONE_OPTIMIZATION_TECHNIQUE">SPEED</PARAM>
    <PARAM name="ADV_NETLIST_OPT_SYNTH_GATE_RETIME">ON</PARAM>
    <PARAM name="ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP">ON</PARAM>
    <PARAM name="STRATIX_OPTIMIZATION_TECHNIQUE">SPEED</PARAM>
 </POINT>
    <POINT space="fit" enabled="1">
  <PARAM name="PHYSICAL_SYNTHESIS_REGISTER_RETIMING">ON</PARAM>
  <PARAM name="PHYSICAL_SYNTHESIS_REGISTER_DUPLICATION">
   ON</PARAM>
  <PARAM name="AUTO_PACKED_REG_CYCLONE">OFF</PARAM>
  <PARAM name="AUTO_PACKED_REGISTERS_STRATIX">OFF</PARAM>
  <PARAM name="SEED">3</PARAM>
  <PARAM name="PHYSICAL_SYNTHESIS_COMBO_LOGIC">ON</PARAM>
        </POINT>
</DESIGNSPACE>
```

The example defines a custom exploration space that has two points: one map exploration point which changes synthesis settings, and one fit exploration point which change the Quartus II Fitter settings. The map point sets the optimization technique to speed, turns on gate-level retiming, and turns on the WYSIWYG resynthesis. For the fit point, register retiming, register duplication, and physical synthesis for combinational logic are turned on; register packing is turned off; and a seed value of three is used.

Custom Space XML Schema

Example 12–4 contains an XML Schema describing the XML format for custom exploration space files. You can use an advanced XML editor or XML verification tool to validate any custom exploration files against this schema.

Example 12–4. XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:element name="DESIGNSPACE">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="COPYRIGHT" type="xs:string" minOccurs="0"/>
          <xs:element name="POINT" maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="PARAM" minOccurs="0" maxOccurs="unbounded">
                  <xs:complexType>
                    <xs:simpleContent>
                      <xs:extension base="xs:string">
                        <xs:attribute name="name" type="xs:string" use="required"/>
                      </xs:extension>
                    </xs:simpleContent>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
             <xs:attribute name="space" type="xs:string" use="required"/>
             <xs:attribute name="enabled" type="xs:boolean" use="optional" default="1"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="optional"/>
    </xs:complexType>
  </xs:element:
</xs:schema
```

Referenced Documents

This chapter references the following documents:

- *Netlist Optimizations & Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*

Document Revision History

Table 12–9 shows the revision history for this chapter.

Table 12–9. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 12–27.	—
May 2007 v7.1.0	• Added Arria GX to list of devices that support all Advanced setting types • Removed device-centric specificity in Table 12–7 and Table 12–8 to cover a broader range of devices (device families) • Added Referenced Documents on page 12–27.	Added support information for the Arria GX device.
March 2007 v7.0.0	Updates for software version 7.0, including: • Made minor changes to DSE Support for Altera Device Families • Made minor changes to Distribute DSE Using LSF Resources • Updated XML Schema (Example 12-4)	Added support information for the Cyclone III device.
November 2006 v6.1.0	Added revision history to the document	Added support information for the Stratix III device.
May 2006 v6.0.0	Updated text for the software version 6.0.0, including information about the new Quartus II TimeQuest Timing Analyzer feature	—
October 2005 v5.1.0	Chapter 12 was formerly Chapter 10 in version 5.0	—
May 2005 v5.0.0	Chapter 10 was formerly Chapter 9 in version 4.2	—
December 2004 v2.1	Updated to reflect the new functionality in software version 4.1, including tables and figures	—
June 2004 v2.0	Updated to reflect the new functionality in software version 4.0, including tables and figures	—
February 2004 v1.0	Initial release	—

13. Synplicity Amplify Physical Synthesis Support

QII52011-7.2.0

Introduction

Synplicity has developed the Amplify Physical Optimizer physical synthesis software to help designers meet performance and time-to-market goals. You can use this software to create location assignments and optimize critical paths outside the Quartus® II software design environment. The Amplify Physical Optimizer design software, which runs on the Synplify Pro synthesis engine, creates a Tcl script with hard location assignments and LogicLock™ regions to control logic placement in the Quartus II software. Depending on the design, the Amplify Physical Optimizer software can improve Altera® device performance over Synplify Pro-compiled designs by reducing the number of logic levels and the interconnect delays in critical paths. Moreover, the Amplify Physical Optimizer software allows designers to compile multiple implementations in parallel to reduce optimization time.



For more information on the Synplify Pro software, refer to *Synplicity Synplify & SynplifyPro Support* chapter in volume 1 of the *Quartus II Handbook*.

This chapter explains the physical synthesis concepts, including an overview of the Amplify Physical Optimizer software and Quartus II flow.

Software Requirements

The examples in this document were generated using the following software versions:

- Quartus II, version 5.1
- Amplify Physical Optimizer, version 3.7

Amplify Physical Synthesis Concepts

The Amplify Physical Optimizer physical synthesis tool uses information about the interconnect architectures of Altera devices to reduce interconnect and logic delays in the critical paths. Timing-driven synthesis tools cannot accurately predict how place-and-route tools function; therefore, determining the real critical path with the synthesis tool is a difficult task.

Synthesis tools create technology-level netlist files that work with floorplans using place-and-route tools. Synthesis tools also define netlist names that are used in place-and-route, which means hard location assignments may not apply in the next revision of the resynthesized netlist as nodes names might have been renamed or removed.

Physical synthesis allows you to create floorplans at the register transfer level (RTL) of a design, giving you the ability to perform logic tunneling and replication. Physical synthesis also gives you the flexibility to make changes at the RTL level, allowing these changes to reflect in previously planned paths.

Physical synthesis uses knowledge of the FPGA device architecture to place paths into customized regions. This process will minimize interconnect delays as interconnect and placement information influences the synthesis process of the design.

When the Amplify Physical Optimizer software synthesizes a design, it creates a .**vqm** atom-netlist and Tcl script files, which are read by the Quartus II software. You can create a Quartus II project with the VQM netlist as the top-level module and source the Tcl script generated by the Amplify Physical Optimizer software. The Tcl script sets the design's device, timing constraints (Timing Driven Compilation [TDC] value, multicycle paths, and false paths), and any other constraints specified by the Amplify Physical Optimizer software. After you source the Tcl script, you can compile the design in the Quartus II software.

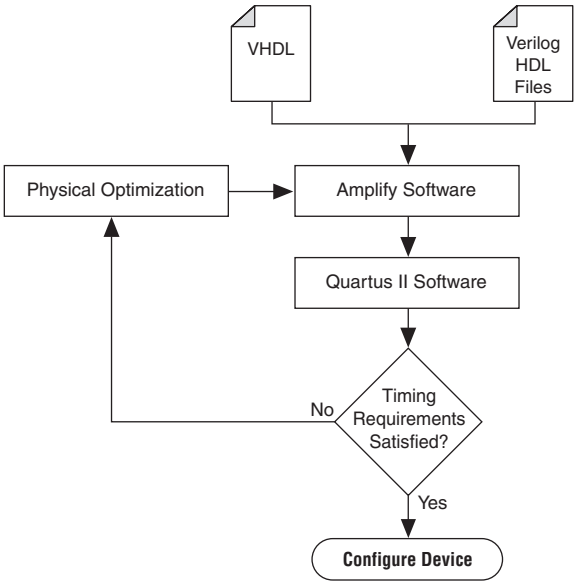
Refer to "Forward Annotating Amplify Physical Optimizer Constraints into the Quartus II Software" on page 13–12 for more information on setting up a Quartus II project with Amplify Physical Optimizer Tcl script files.

After the Quartus II software compiles the design, the software performs a timing analysis on the design. The timing analysis reports all timing-related information for the design. If the design does not meet the timing requirements, you can use the timing analysis numbers as a reference when running the next iteration of physical synthesis through the Amplify Physical Optimizer software. This same timing analysis information is also reported in a file called <*revision name*>**.tan.rpt** in the design directory.

Amplify-to-Quartus II Flow

If timing requirements are not met with the Amplify Physical Optimizer flow, you should first place and route the design in the Quartus II software without physical constraints. After compilation, you can determine which critical paths should be optimized in the Amplify Physical Optimizer tool in the next iteration. Figure 13–1 shows the Amplify Physical Optimizer design flow.

Figure 13–1. Software Design Flow



Initial Pass: No Physical Constraints

The initial iteration involves synthesizing the design in the Amplify Physical Optimizer software without physical constraints.

Before beginning the physical synthesis flow, run an initial pass in the Amplify Physical Optimizer without physical constraints. At the completion of every Quartus II compilation, the Quartus II Timing Analyzer performs a comprehensive static timing analysis on your design and reports your design's performance and any timing violations. If the design does not meet performance requirements after the first pass, additional passes can be made in the Amplify software.

Create New Implementations

To set the Amplify Physical Optimizer software options, perform the following steps:

1. Compile the design with the **Resource Sharing** and **FSM Compiler** options selected and the **Frequency** setting specified in MHz. For optimal synthesis, the Amplify software includes the retiming, pipelining, and FSM Explorer options. For designs with multiple clocks, set the frequency of individual clocks with Synthesis Constraints Optimization Environment (SCOPE).

2. Select **New Implementation**. The **Options for Implementation** dialog box appears.

3. Specify the part, package, and speed grade of the targeted device in the **Device** tab.

4. Turn on the **Map Logic to Atoms** option in the **Device Mapping Options** dialog box.

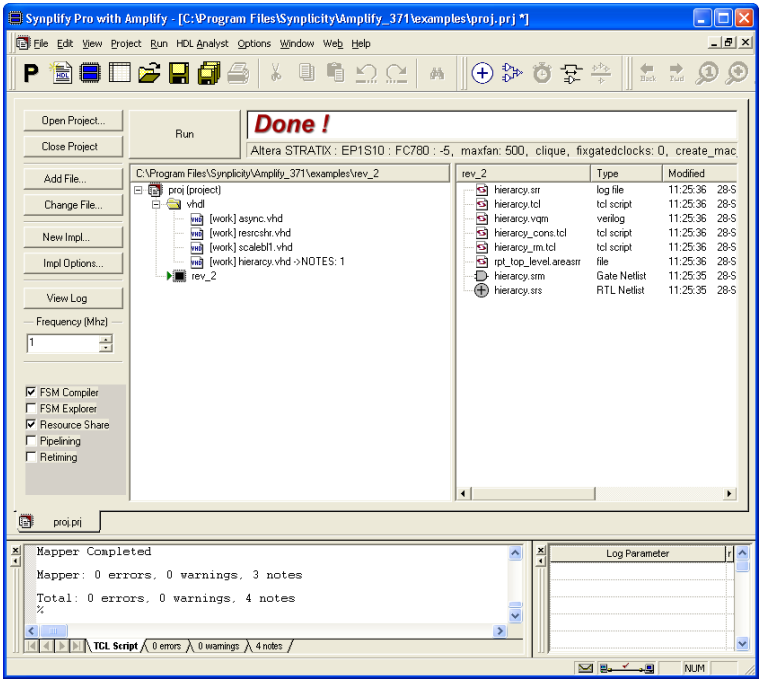
5. Turn off the **Disable I/O Insertion** and **Perform Cliquing** options.

6. Specify the name and directory in the **Implementation Results** tab. The result format should be VQM, and you should select **Optional Output Files** as the **Write Vendor Constraint File** option so that the software can generate the Tcl script containing the project constraints.

7. Specify the number of critical paths and the number of start and end points to report in the **Timing Report** tab. Figure 13–2 shows the main Amplify Physical Optimizer project window.

These steps create a directory where the results of this pass are recorded. Ensure that the Amplify Physical Optimizer software implementation options are set as described in the initial pass.

Figure 13–2. Amplify Physical Optimizer Project Window



Iterative Passes: Optimizing the Critical Paths

In the iterative passes, you optimize the design by placing logic in the device floorplan within the Amplify software. Amplify's floorplan is a high-level view of the device architecture. The floorplan view is dependent upon the target device family. When the Amplify Physical Optimizer re-optimizes the current critical path, additional critical paths may be created. Continue to add new constraints to the existing floorplan until it meets the performance requirements. The design may need several iterations to meet these performance requirements. Since optimizing critical paths involves trying different implementations, the creation of various Amplify project implementations will help in organizing the placement of logic in the floorplan.

Using the Amplify Physical Optimizer Floorplans

When designs do not meet performance requirements with the initial pass through the Amplify Physical Optimizer software, you can create location assignments to reduce interconnect and logic delays to improve your design's performance.

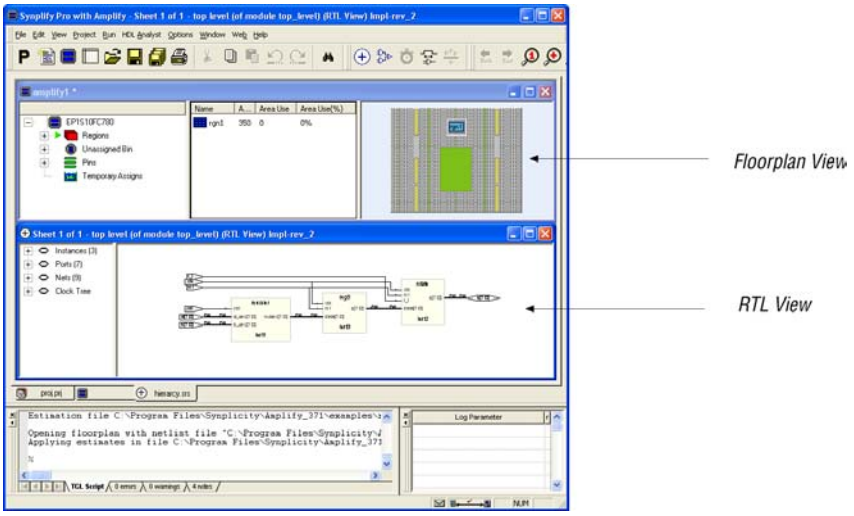
You must determine which paths to constrain based on the critical paths from the previous implementation. When Quartus II projects are launched with the Amplify Tcl script, the Quartus II software generates a **<***revision name***>.tan. rpt** file that lists the critical paths for the design. You can then create custom structure regions for critical paths. After critical paths are implemented in a floorplan with the Amplify Physical Optimizer software, you must resynthesize the design. The software will then attempt to optimize the critical paths and reduce the number of logic levels. After the Amplify Physical Optimizer software resynthesizes the design, the Quartus II software must compile the new implementation. If the design does not meet timing requirements, perform another physical synthesis iteration.

Use the following steps to create a floorplan in the Amplify Physical Optimizer software:

1. Click the **New Physical Constraint File** icon at the top of the Amplify Physical Optimizer window.

2. Click **Yes** on the **Estimation Needed** dialog box; the floorplan window appears (Figure 13–3).

Figure 13–3. Stratix 1S20 Floorplan in the Amplify Physical Optimizer Software



The floorplan view is located at the top of the screen and the RTL view is at the bottom of the screen.

You can specify modules or individual paths in the Amplify Physical Optimizer software. Using modules can quickly resolve timing problems.

Use the following steps in the software to create a floorplan module:

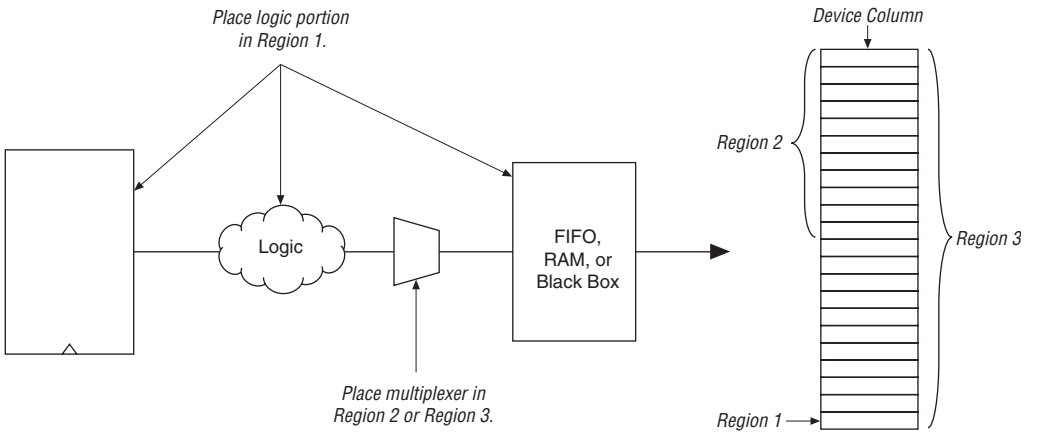
1. Create a region in the Amplify Physical Optimizer device floorplan window and select the module in the RTL view of the design.
2. Drag the module to the new region. The software will then report the utilization of the region.
3. Resynthesize the design in the software to reoptimize the critical path after the modules have location constraints.
4. Write out the placement constraints into the VQM netlist and the Tcl script.

Repeat the above procedure to create as many regions as required.

Multiplexers

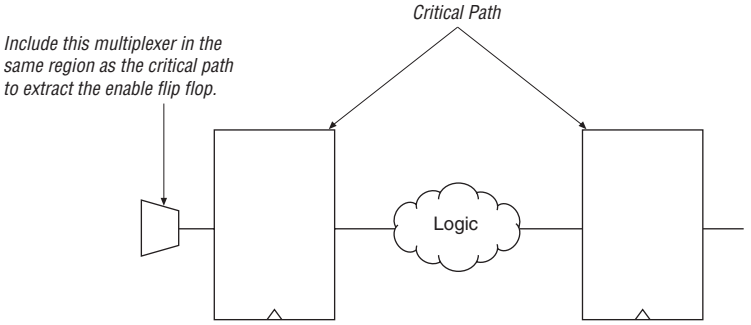
To create a floorplan for critical paths with one or more multiplexers, create multiple regions and assign the multiplexer to one region and the logic to another. Figure 13–4 shows placing critical paths with multiplexers.

Figure 13–4. Placing Critical Paths with Multiplexers



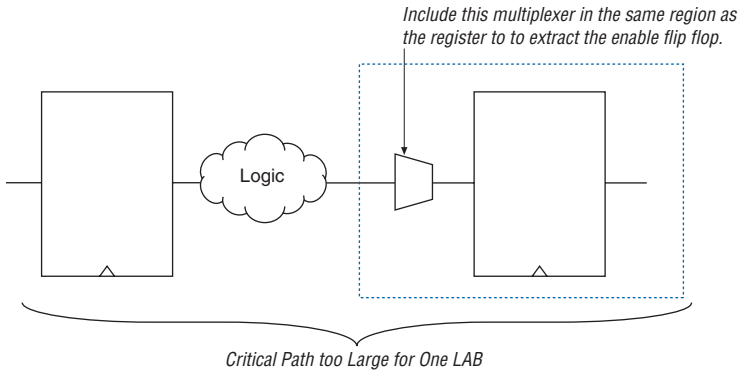
If the critical path contains a multiplexer feeding a register, create a region and place the multiplexer along with the entire critical path in the region (Figure 13–5).

Figure 13–5. Critical Paths with Multiplexers Feeding Registers



If the critical path is too large for the region, divide the critical path and ensure that the multiplexer and register are in the same region. Figure 13–6 shows large critical paths with multiplexers feeding registers.

Figure 13–6. Large Critical Paths with Multiplexers Feeding Registers



Independent Paths

Designs may have two or more independent critical paths. To create an independent path in the Amplify Physical Optimizer software, follow the steps below:

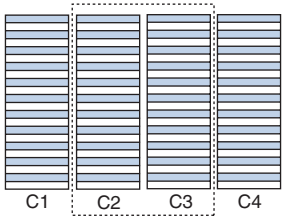
1. Create a region and assign the first critical path to that region.
2. Create another region, leaving one MegaLAB structure between the first and second regions.
3. Assign the second critical path to the second region.

Feedback Paths

If critical paths have the same start and end points, follow the steps below in the Amplify Physical Optimizer software (Figure 13–7):

1. Select the register and instance not directly connected to the register.
2. Right-click and select **Filter Schematic** twice.
3. Highlight the line leading out of the register and either press **P** or right-click the line. Select **Expand Paths**. Assign this logic to a region.

Figure 13–7. Critical Paths with the Same Starting or Ending Points

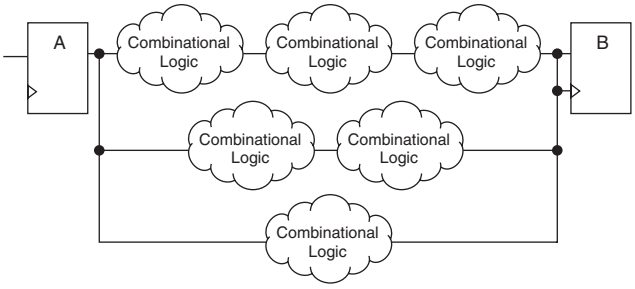


Starting and Ending Points

Figure 13–8 shows a critical path that has multiple starting and ending points. Use **Find** to display all the starting and ending points in the RTL view in Amplify. Expand the paths between those points. If there is unrelated logic between the multiple starting points and ending points,

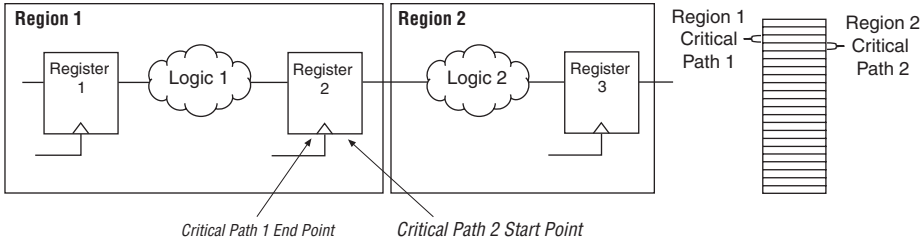
assign the starting points and ending points to the same region. Similarly, if there is unrelated logic between starting points and multiple ending points, assign the starting points and ending points to the same region.

Figure 13–8. Critical Paths with Multiple Starting or Ending Points



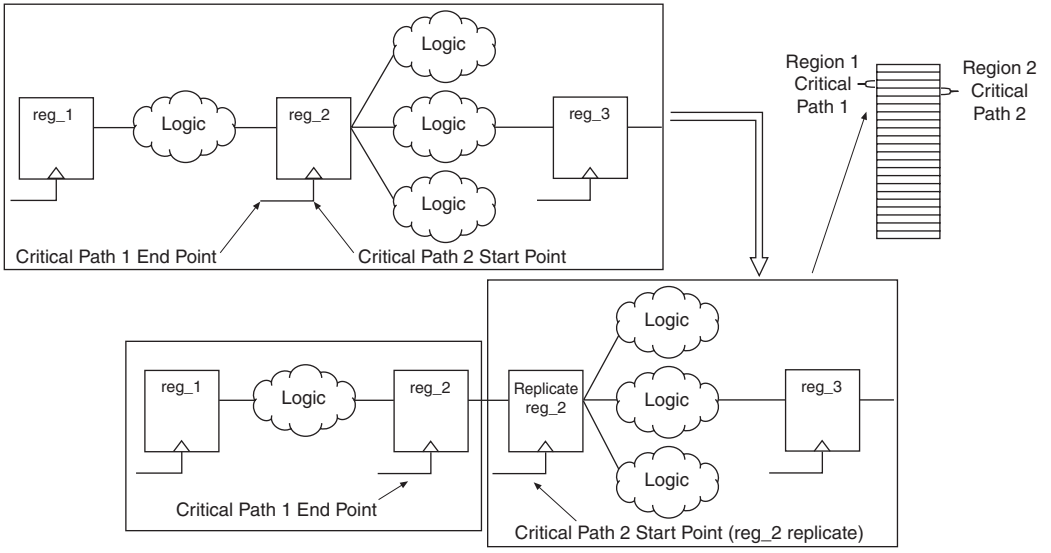
If the two critical paths share a register at the starting or ending point, assign one critical path to one region, and assign the other critical path to an adjacent region. Figure 13–9 shows two critical paths that share a register.

Figure 13–9. Two Critical Paths Sharing a Register



If the fanout is on the shared region, replicate the register and assign both registers to two regions (Figure 13–10). This is done by dragging the same register to the required regions. Entities and nodes are also replicated by performing the same procedure.

Figure 13–10. Fanout on a Shared Region



Utilization

Designs with device utilizations of 90% or higher may have difficulties during fitting in the Quartus II software. If the device has several finite state machines, you should implement the state machines with sequential encoding, as opposed to one-hot encoding.

To check area utilization, check the Amplify Physical Optimizer **log** file and **.srr** file for region utilization, after the mapping stage is complete. On the Run menu, click **Estimate Area** to update the utilization estimates.

Detailed Floorplans

If the critical path does not meet timing requirements after physical optimization, you can create new regions to achieve timing closure. It is recommended that regions do not overlap. Regions should either be entirely contained in another region or remain entirely outside of it. Select the logic requiring optimization from the existing region. Deselect the logic and assign it to the new region. Run the Amplify Physical Optimizer software on the design with the modified physical constraints. Then place and route the design.

Forward Annotating Amplify Physical Optimizer Constraints into the Quartus II Software

The Amplify Physical Optimizer software simplifies the forward annotating of both timing and location constraints into the Quartus II software through the generation of three Tcl scripts. At the completion of a physical synthesis run, in the Amplify Physical Optimizer software, the following Tcl scripts are generated:

- <*project name*>**_cons.tcl**
- <*project name*>**.tcl**
- <*project name*>**_rm.tcl**

Table 13–1 provides a description of each script's purpose.

Table 13–1. Amplify Physical Optimizer Tcl Script Description

Tcl File	Description
<*project name*>**_cons**	This Tcl script will create and compile a Quartus II project. The <*project name*>**.tcl** will automatically be sourced when this script is sourced.
<*project name*>	This script contains forward annotation of constraint information including clock frequency, duty cycle, location, etc.
<*project name*>**_rm**	This script removes any previous constraints from the project. The removed constrainst is saved in <*project name*>**_prev.tcl**

To forward annotate Amplify Physical Optimizer's constraints into the Quartus II software you must use **quartus_cmd**. The **quartus_cmd** command must be used as Amplify Physical Optimizer's Tcl scripts are not compatible with **quartus_sh**. The following command will execute the <*project name*>**_cons**, which will create a Quartus II project with all Amplify Physical Optimizer constraints forward annotated, and will perform a compilation.

```
<command prompt>quartus_cmd f-my_project_cons.tcl ↵
```

You must run the <*project name*>**_cons.tcl** first.

After compilation, you may customize the project either in the Quartus II GUI or sourcing a custom Tcl script.

Refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook* for more information on creating and understanding Tcl scripts in the Quartus II software.

Altera Megafunctions Using the MegaWizard Plug-In Manager with the Amplify Software

When you use the Quartus II MegaWizard® Plug-In Manager to set up and parameterize a megafunction, it creates either a VHDL or Verilog HDL wrapper file. This file instantiates the megafunction (a black box methodology) or, for some megafunctions, generates a fully synthesizeable netlist for improved results with EDA synthesis tools such as Synplify (a clear box methodology).

Clear Box Methodology

The MegaWizard Plug-In Manager-generated fully synthesizeable netlist is referred to as a clear box methodology because the Amplify Physical Optimizer software can "see" into the megafunction file. The clear box feature enables the synthesis tool to report more accurate timing estimates and take better advantage of timing driven optimization.

To turn on the clear box, go to the Tools menu, and select the **MegaWizard Plug-In Manager.** Turn on the **Generate Clearbox body (for EDA tools only)** option. This option is only for certain megafunctions. If this option does not appear, then clear box models are not supported for the selected megafunction. Turning on this option causes the MegaWizard Plug-In Manager to generate a synthesizable clear box netlist instead of the megafunction wrapper file described in "Black Box Methodology" on page 13–14.

Using MegaWizard Plug-In Manager-generated Verilog HDL Files for Clear Box Megafunction Instantiation

If you check the <*output file*>**_inst.v** option on the last page of the wizard, the MegaWizard Plug-In Manager generates a Verilog HDL instantiation template file for use in your Synplify design. This file can help you instantiate the megafunction clear box netlist file, <*output file*>**.v**, in your top-level design. Include the megafunction clear box netlist file in your Amplify Physical Optimizer project and the information gets passed to the Quartus II software in the Amplify Physical Optimizer-generated VQM output file.

Using MegaWizard Plug-In Manager-generated VHDL Files for Clear Box Megafunction Instantiation

If you check the <*output file*>.**cmp** and <*output file*>**_inst.vhd** options on the last page of the wizard, the MegaWizard Plug-In Manager generates a VHDL component declaration file and a VHDL instantiation template file for use in your design. These files help to instantiate the megafunction clear box netlist file, <*output file*>.**vhd**, in your top-level design. Include

the megafunction clear box netlist file in your Amplify Physical Optimizer project and the information gets passed to the Quartus II software in the Amplify Physical Optimizer-generated VQM output file.

Black Box Methodology

The MegaWizard Plug-In Manager-generated wrapper file is referred to as a black-box methodology because the megafunction is treated as a "black box" in the Amplify Physical Optimizer software. The black box wrapper file is generated by default in the MegaWizard Plug-In Manager and is available for all megafunctions.

The black-box methodology does not allow the synthesis tool any visibility into the function module thus not taking full advantage of the synthesis tool's timing driven optimization. For better timing optimization, especially if the black box does not have registered inputs and outputs, add timing models to black boxes.



For more information on instantiating MegaWizard Plug-In Manager modules or black boxes, refer to the *Synplicity Synplify & SynplifyPro Support* chapter in volume 1 of the *Quartus II Handbook*.

Conclusion

Physical synthesis uses improved delay estimation to optimize critical paths. The Amplify Physical Optimizer software uses the hierarchical structure of logic and interconnect in Altera devices so that designers can direct a critical path to be placed into several well-defined blocks. The Amplify Physical Optimizer-to-Quartus II software flow is one of the steps to solving the problem of achieving timing closure through physical synthesis.

Referenced Documents

This chapter references the following documents:

- *Synplicity Synplify and SynplifyPro Support* chapter in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 13–2 shows the revision history for this chapter.

Table 13–2. Document Revision History

Data and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No changes.	—
May 2007 v7.1.0	Added "Referenced Documents" on page 13–14.	—
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Added revision history to this chapter.	—
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0.	—
October 2005 v5.1.0	Chapter 14 was formerly Chapter 12 in version 5.0.	—
May 2005 v5.0.0	Chapter 12 was formerly Chapter 11 in version 4.2.	—
December 2004 v1.1	• Chapter 11 was formerly Chapter 12. • Updates to tables and figures. • New functionality in the Quartus II software version 4.2	—
February 2004 v1.0	Initial release.	—

Section IV. Engineering Change Management

Programmable logic can accommodate changes to a system specification late in the design cycle. Last-minute design changes, commonly referred to as engineering change orders (ECOs), are small changes to the functionality of a design after the design has been fully compiled. This section describes how the Chip Planner feature in the Quartus II software supports ECOs by allowing quick and efficient changes to your logic late in the design cycle.

This section includes the following chapter:

- Chapter 14, Engineering Change Management with the Chip Planner



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

14. Engineering Change Management with the Chip Planner

QII52017-7.2.0

Introduction

Programmable logic can accommodate changes to a system specification late in the design cycle. In a typical engineering project development cycle, the specification for the programmable logic portion is likely to change after engineering development begins or while integrating all system elements.

Last-minute design changes, commonly referred to as engineering change orders (ECOs), are small changes to the functionality of a design after the design has been fully compiled. A design is fully compiled when synthesis and place-and-route are completed.

The Chip Planner supports ECOs by allowing quick and efficient changes to your logic late in the design cycle. It provides a visual display of your post place-and-route design mapped to the device architecture of your chosen FPGA, from LAB placement in the device to each mapped Logic Element (LE) or Adaptive Logic Module (ALM). You can analyze your design with the visual display to alter how device resources are mapped to support ECOs.

This chapter addresses the impact that ECOs have on the design cycle, discusses the design flow for performing ECOs, and describes how you can use the Chip Planner to perform ECOs.



In addition to performing ECOs, the Chip Planner allows you to perform detailed analysis on routing congestion, relative resource usage, logic placement, LogicLock™ and customized regions, fan-ins and fan-outs, paths between registers, and delay estimates for paths.



For detailed information about using the Chip Planner for design analysis, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

The Chip Planner supports the following device families:

- Arria™ GX
- Stratix® III
- Stratix II
- Stratix II GX
- Stratix
- Stratix GX

- HardCopy® II
- HardCopy
- Cyclone® III
- Cyclone II
- Cyclone
- MAX II®

Engineering Change Orders

ECOs are typically performed during the verification stage of a design cycle. When a small change is required on a design, such as modifying a PLL for a different clock frequency or routing a signal out to a pin for analysis, recompilation of the entire design can be time consuming, especially for larger designs. Because several iterations of small design changes can occur during the verification cycle, recompilation times can quickly add up. Furthermore, a full recompilation due to a small design change can result in the loss of previous design optimizations. Performing ECOs, instead of performing a full recompilation on your design, limits the change only to the affected portions of logic.

This section discusses the areas in which ECOs have an impact on a system design and how the Quartus II software can help you optimize the design in these areas. The following topics are discussed in this section:

- "Performance"
- "Compilation Time" on page 14–3
- "Verification" on page 14–3
- "Documentation" on page 14–4

Performance

Making a small change to the design functionality can result in a loss of previous design optimizations. Typical examples of design optimizations are floorplan optimizations and physical synthesis. Ideally, you should preserve previous design optimizations.

The Chip Planner allows you to perform ECOs directly on the post place-and-route database of your design. Any changes you make are restricted to the affected device resources, so the timing performance of the remaining portions of your design are not affected. The Chip Planner performs design rule checks on all changes to prevent illegal modifications to your design.

Additionally, the Quartus II software offers an incremental compilation feature that preserves the optimizations and placement of your design during recompilation. This feature allows you to create partitions of your

design, so that if a change is required after the design is fully placed and optimized, only the affected partition is recompiled to implement the change.

The incremental compilation flow fully supports performing ECOs with the Chip Planner.

When recompiling a project with the Quartus II incremental compilation enabled, the compiler preserves all ECOs performed with the Chip Planner in partitions that have not been modified.



For more information about how to use the incremental compilation feature in the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

For more information about using the ECO flow in conjunction with incremental compilation, refer to "Using Incremental Compilation in the ECO Flow" on page 14–38.

Compilation Time

In the traditional programmable logic design flow, a small change in the design requires a complete recompilation of the design. A complete recompilation of the design consists of synthesis and place-and-route. Making small changes to the design to reach the final implementation on a board can be a long process. Since the Chip Planner works only on the post place-and-route database, you can implement your design changes in minutes without performing a full compilation.

Verification

After you make a design change, you can verify the impact to your design. To verify that you have not violated timing, you can perform a static timing analysis using the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer after you check and save your netlist changes within the Chip Planner.



For more information about the Quartus II TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*. For more information about the Quartus II Classic Timing Analyzer, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Additionally, you can perform a gate-level or timing simulation of the ECO-modified design by using the post place-and-route netlist generated by the Quartus II software.

Documentation

All ECOs made with the Chip Planner are logged in the Change Manager to provide a track record of all changes. By using the Change Manager, you can easily revert back to the original post-fit netlist or you can pick and choose which ECOs you want to have applied.

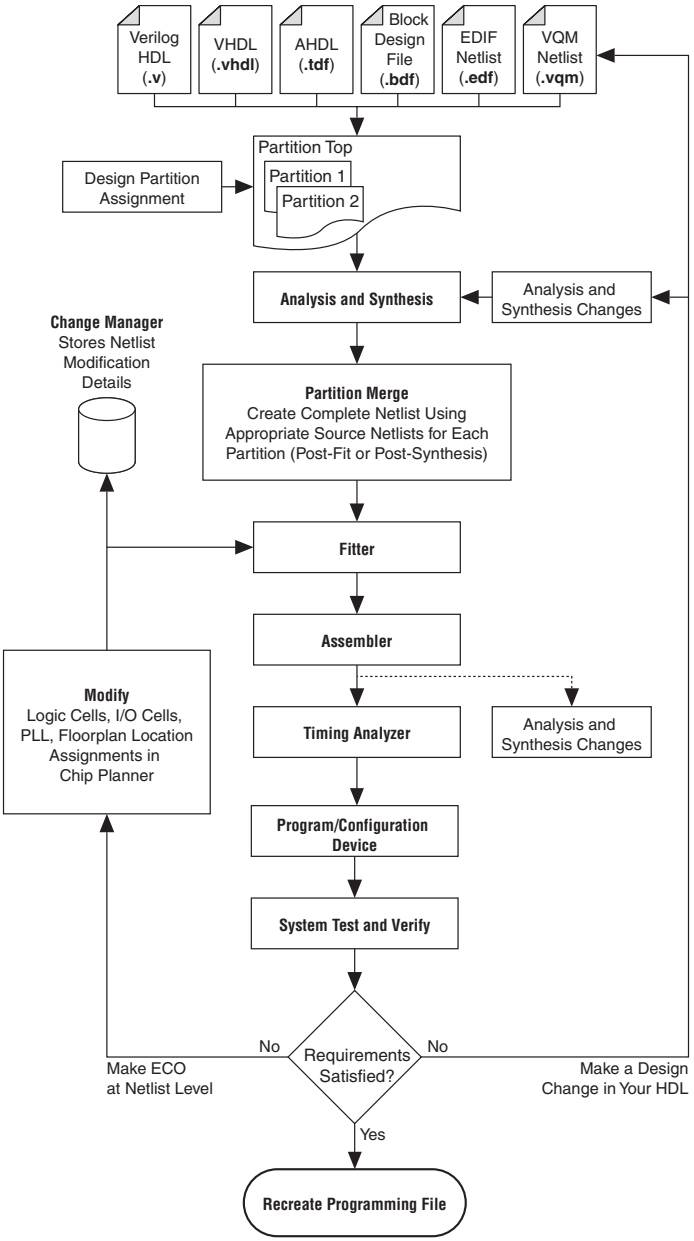
For more information about the Change Manager, refer to "Change Manager" on page 14–36.

Additionally, the Quartus II software provides support for multiple compilation revisions of the same project. You can use ECOs made with the Chip Planner in conjunction with revision support to compare several different ECO changes and to provide the ability to revert back to previous project revisions.

ECO Design Flow

Figure 14–1 shows the design flow for performing ECOs.

Figure 14–1. Design Flow to Support ECO Changes



For iterative verification cycles, implementing small design changes at the netlist level can be faster than making an RTL code change. As such, making ECO changes are especially helpful when you debug the design on silicon and need a fast turnaround to generate a programming file for debugging the system.

A typical ECO application occurs when you uncover a problem on the board and isolate the problem to the appropriate nodes or I/O cells on the device. You must be able to correct the functionality quickly and generate a new programming file. Performing small changes using the Chip Planner allows you to modify the post place-and-route netlist directly. This bypasses the need to perform synthesis and logic mapping, thus decreasing the turn around time for programming file generation during the verification cycle. If the change corrects the problem, no modification of the HDL source code is necessary. You can use the Chip Planner to perform the following ECO-related changes to your design:

- Document the changes made with the Change Manager
- Easily recreate the steps taken to produce design changes
- Generate EDA simulation netlists for design verification
- Perform static timing analysis on the design



The Quartus II software can help reduce recompilation time with incremental recompilation for more complex changes that require HDL source code modifications.

The Chip Planner Overview

The Chip Planner provides a visual display of device resources. It shows the arrangement and usage of the resource atoms in the device architecture that you are targeting. Resource atoms are the building blocks for your device, such as ALMs, LEs, PLLS, DSP blocks, memory block, or IOEs.

The Chip Planner also provides an integrated platform for design analysis and for making ECOs to your design after place-and-route. The toolset consists of the Chip Planner (providing a device floorplan view of your mapped design) and two integrated subtools —the Resource Property Editor and the Change Manager.

For analysis, the Chip Planner can show logic placement, LogicLock and custom regions, relative resource usage, detailed routing information, routing congestion, fan-ins and fan-outs, paths between registers, and delay estimates for paths. Additionally, the Chip Planner allows you to create location constraints or resource assignment changes, such as moving or deleting logic cells or I/O atoms using the device floorplan. For ECO changes, the Chip Planner enables you to create, move, or delete logic cells in the post place-and-route netlist for fast programming file generation. Additionally, you can open the Resource Property Editor

from the Chip Planner to edit the properties of resource atoms or to edit the connections between them. All changes to resource atoms and connections are logged automatically with the Change Manager.

Opening the Chip Planner

To open the Chip Planner, on the Tools menu, click **Chip Planner**. Alternatively, click the **Chip Planner** icon on the Quartus II software toolbar.

Optionally, the Quartus II software supports cross-probing to open the Chip Planner. To open the Chip Planner by cross-probing, use the shortcut menu in the following tools:

- Compilation Report
- Project Navigator window
- RTL source code
- Timing Closure Floorplan
- Node Finder
- Simulation Report
- RTL Viewer



If the device in your project is not supported by the Chip Planner and you attempt to open the Chip Planner, the Quartus II software displays the following message: "Can't display Chip Planner: the current device family is unsupported." In such cases, use the Timing Closure Floorplan.

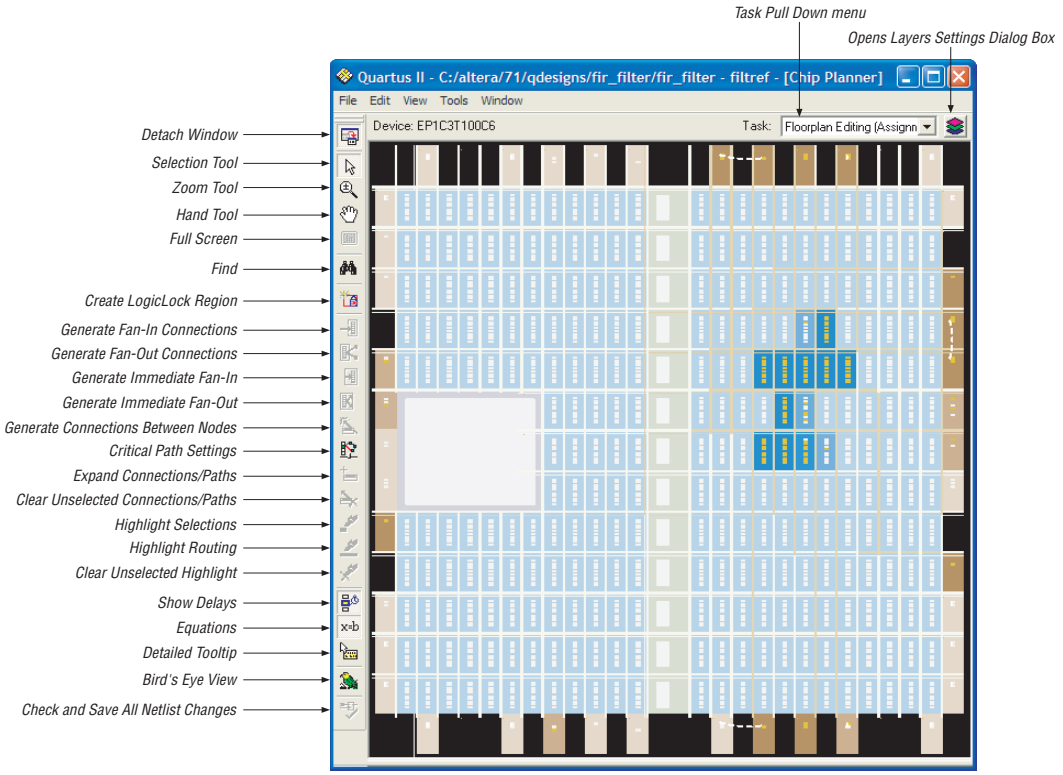


For more information about the Timing Closure Floorplan, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

The Chip Planner Toolbar

The Chip Planner gives you design analysis capabilities with a user-friendly GUI. You can perform many functions within the Chip Planner from the menu items or by clicking the icons on the toolbar. Figure 14–2 shows an example of the Chip Planner toolbar and describes the commonly used icons.

Figure 14–2. Chip Planner Toolbar



You can also customize the icons on the Chip Planner toolbar. To customize the icon toolbar if the Chip Planner window is attached, on the Tools menu, click **Customize Chip Planner**. If the Chip Planner window is detached, on the Tools menu, click **Customize**.

For more information about using the Chip Planner for analyzing your design, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

The Chip Planner Tasks and Layers

The Chip Planner enables you to set up tasks to quickly implement ECO changes or manipulate assignments for the floorplan of the device. Each task consists of an editing mode and a set of customized layer settings.

The editing modes available in the Chip Planner are the Assignment mode and the ECO mode. Assignment mode enables you to create or manipulate LogicLock regions and make location constraints on the resource atoms used in your design. Assignments made are reflected in the Quartus II Settings File (**.qsf)** and the Assignment Editor. With ECO mode, you can create atoms, delete atoms, and move existing atoms to different locations. The changes made with ECO mode are made in the post place-and-route database. You can analyze your design with both modes.

The layers settings enable you to specify the displayed graphic elements for a given task. You can turn off the display of specific graphic elements to increase the window refresh speed and reduce visual clutter when viewing complex designs. The Background Color Map indicates the relative level of resource usage for different areas of the device. For example, **Routing Utilization** indicates the relative routing utilization and **Physical Timing Estimate** indicates the relative physical timing.

The Chip Planner has predefined tasks that enable you to quickly implement ECO changes or manipulate assignments for the floorplan of the device. The Chip Planner provides the following predefined tasks:

- Post-Compilation Editing (ECO)
- Floorplan Editing (Assignment)
- Power Analysis (Assignment)—available for Stratix III devices only

You can choose the predefined task by selecting it in the **Task** pull down menu located in the upper right corner of the Chip Planner floorplan view.

To customize your own task, click on the layers icon to open the **Layers Settings** dialog box.



For more information about assignments and analysis with the Chip Planner, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

For more information about performing ECOs using the ECO mode, refer to "Performing ECOs with the Chip Planner (Floorplan View)" on page 14–15.

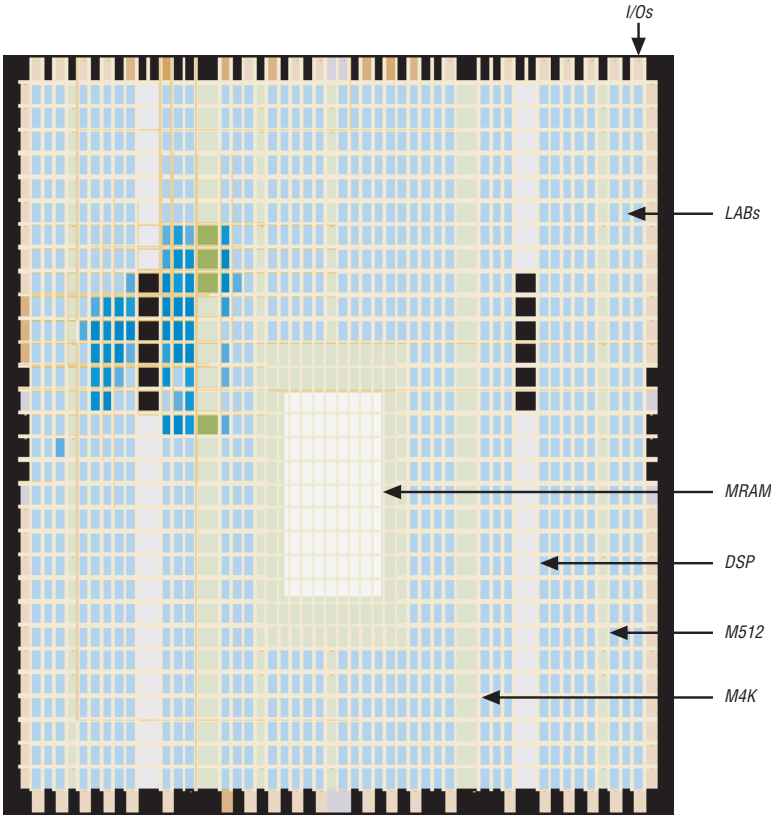
The Chip Planner Floorplan Views

The Chip Planner uses a hierarchical zoom viewer that shows various abstraction levels of the targeted Altera device. As you increase the zoom level, the level of abstraction decreases, thus revealing more detail about your design.

First-Level View

The first zoom level provides a high-level view of the entire device floorplan. This view provides a level of detail similar to the Field View in the Quartus II Timing Closure Floorplan. You can locate and view the placement of any node in your design. Figure 14–3 shows the Chip Planner Floorplan first-level view of a Stratix device.

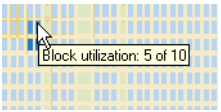
Figure 14–3. The Chip Planner First-Level (Highest) Floorplan View (Stratix Family Device)



Each resource is shown in a different color to help you distinguish between resources. The Chip Planner Floorplan uses a gradient color scheme in which the color becomes darker as the utilization of a resource increases. For example, as more LEs are used in the LAB, the color of the LAB becomes darker.

When you place the mouse pointer over a resource at this level, a tooltip appears that describes, at a high level, the utilization of the resource (Figure 14–4).

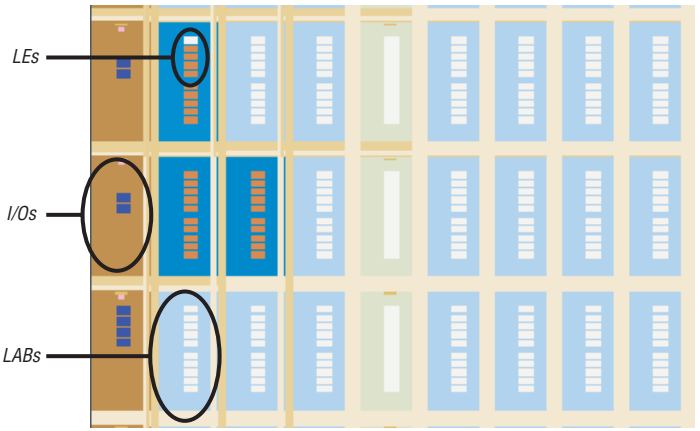
Figure 14–4. Tooltip Message: First-Level View



Second-Level View

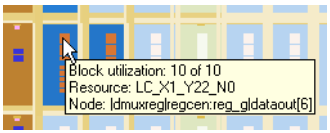
As you zoom in, you see an increase in the level of detail. Figure 14–5 shows the second-level view of the Chip Planner Floorplan for a Stratix device.

Figure 14–5. The Chip Planner Second-Level Floorplan View (Stratix Family Device)



At this level you can see the contents of LABs and I/O banks. You also see the routing channels that are used to connect resources. When you place the mouse pointer over an LE or ALM at this level, a tooltip is displayed (Figure 14–6) that shows the name of the LE/ALM, the location of the LE/ALM, and the number of resources that are used with that LAB. When you place the mouse pointer over an interconnect, the tooltip shows the routing channels that are used by that interconnect.

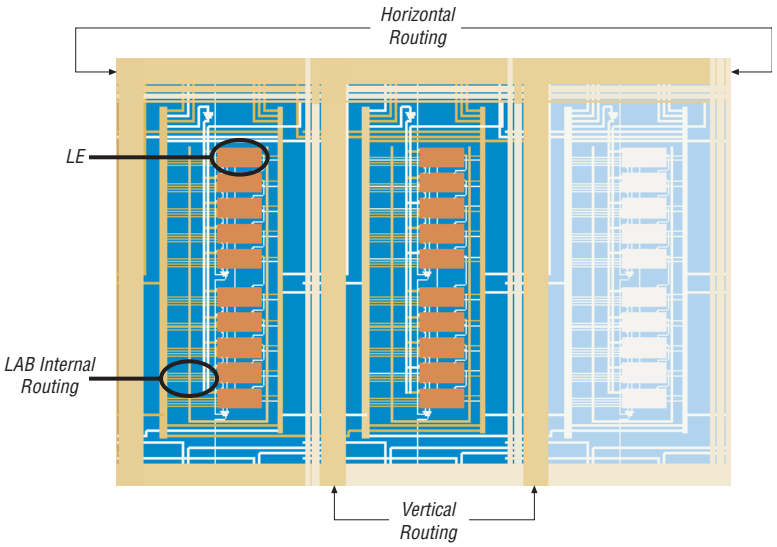
Figure 14–6. Tooltip Message: Second-Level View



Third-Level View

At the third level, which provides the greatest level of detail, you can see each routing resource that is used within a LAB in the FPGA. Figure 14–7 shows the level of detail at the third-level view for a Stratix device.

Figure 14–7. The Chip Planner Third-Level View



The second and third level of zoom allows you to move LEs, ALMs, and I/Os from one physical location to another. You can move a resource by selecting, dragging, and dropping it into the desired location. At this level you can also create new LEs and I/Os.

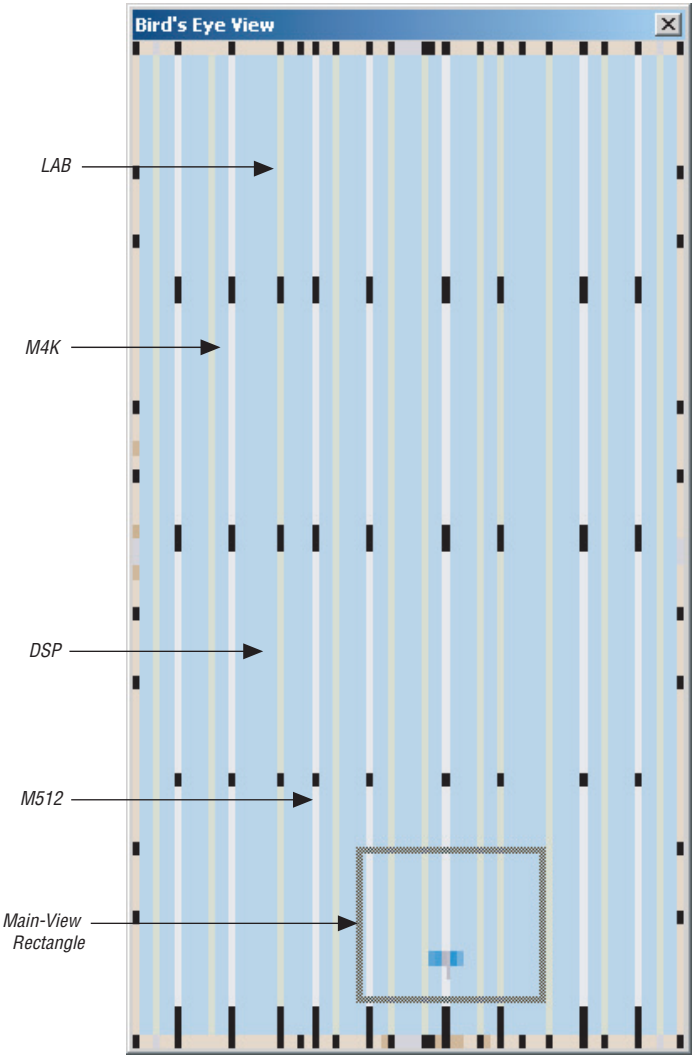
For more information about creating atoms, deleting atoms, or reallocating device atoms, refer to the section "Performing ECOs with the Chip Planner (Floorplan View)" on page 14–15.

For more information about creating Floorplan Assignments, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Bird's Eye View

The Bird's Eye View (Figure 14–8) displays a high-level picture of resource usage for the entire chip and provides a fast and efficient way to navigate between areas of interest in the Chip Planner.

Figure 14–8. Bird's Eye View



The Bird's Eye View is displayed as a separate window that is linked to the Chip Planner Floorplan. When you select an area of interest in the Bird's Eye View, the Chip Planner Floorplan automatically refreshes to show that region of the device. As you change the size of the main-view rectangle in the Bird's Eye View window, the main Chip Planner

Floorplan window also zooms in (or zooms out). You can make the main-view rectangle smaller in the Bird's Eye View to see more detail on the Chip Planner Floorplan window.

The Bird's Eye View is particularly useful when the parts of your design that you are interested in are at opposite ends of the chip and you want to quickly navigate between resource elements without losing your frame of reference.

Performing ECOs with the Chip Planner (Floorplan View)

You can manipulate resource atoms in the Chip Planner when you select the **ECO** editing mode. The following ECO changes can be performed with the Chip Planner Floorplan view:

- "Creating Atoms"
- "Deleting Atoms" on page 14–20
- "Moving Atoms" on page 14–20

To select the **ECO** editing mode in the Chip Planner, perform the following steps with the Chip Planner open:

1. On the View menu, click **Layers Settings**. The Layers Setting window appears.
2. Under **Editing Mode**, select **ECO**.

Creating Atoms

While in the **ECO** editing mode, the Chip Planner enables you to easily create atoms by moving the mouse pointer over the desired resource atom, right-clicking, and then clicking **Create Atom**. After the atoms are created, the properties can be edited by double-clicking the resource atom, which opens the Resource Property Editor.

The type of atoms that you can create are:

- ALMs (for Arria GX, Stratix III, Stratix II, and Stratix II GX)
- LEs (for Stratix, Stratix GX, Cyclone III, Cyclone II, Cyclone, and MAX II)
- I/O Elements



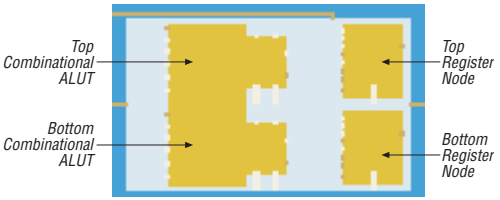
Creating resource atoms is not supported in the **Assignment** editing mode.

Refer to "Resource Property Editor" on page 14–21 for details about editing atom resource properties.

Creating ALM Atoms

Each ALM for Stratix III, Stratix II, and Arria GX device families has two combinational LUT outputs and two registered outputs. In the Chip Planner, you can divide each ALM into four resource atoms according to the type of output path. Figure 14–9 shows an ALM as shown in the Chip Planner.

Figure 14–9. ALM in the Chip Planner



To create a combinational ALM LUT atom, perform the following steps:

1. Right click on the left side of any unused (not shaded) ALM and click **Create Atom**. The **Resource Selection** dialog box appears.
2. In the **Resource Selection** dialog box, select the atom that you wish to create. The lower index number refers to the top combinational node and the higher index refers to the bottom combinational node.
3. Click **OK**. The **Create <*Altera device*> LUT Atom** dialog box appears.
4. In the **Atom Name** box, type the name of the resource atom.
5. Under **LUT Mode**, select from **Normal**, **Extended**, or **Arithmetic**.
6. If applicable, in the **Partition** list, select the partition that the newly created atom should reside in. The default partition for newly created atoms is the top-level partition.
7. Click **OK**.

To find more information about the LUT mode, refer to the data sheet of the appropriate device.

When you have successfully created a combinational output, the combinational element is colored in the Chip Planner. Figure 14–10 shows a combinational ALUT atom.

Figure 14–10. Combinational ALUT Atom



To create a registered ALM atom, perform the following steps:

1. Right click on any ALM register resource and click **Create Atom**. The **Create Register Atom** dialog box appears.
2. In the **Atom Name** box, type the atom name.
3. Click **OK**.

Creating Logic Element Atoms

The Chip Planner shows resource atoms for Stratix, Cyclone, and MAX device families as logic elements. For Cyclone II devices, the Chip Planner shows resource atoms as the combinational output of the logic element LUT and the registered output of the logic element. Figure 14–11 show an example of an atom resource in the Chip Planner for the Stratix, Cyclone, and MAX devices. Figure 14–12 shows the Cyclone II resource atom in the Chip Planner.

Figure 14–11. Logic Element for Stratix, Cyclone, and MAX Devices in the Chip Planner

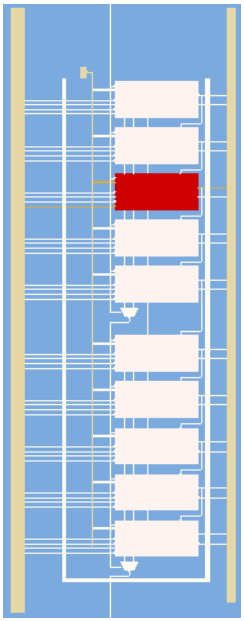
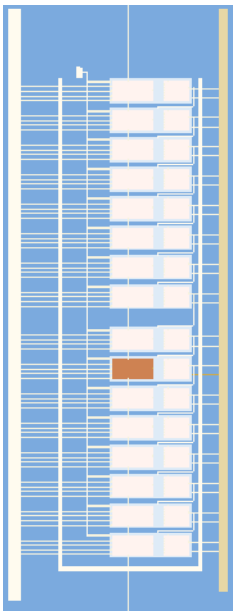


Figure 14–12. Logic Element for Cyclone II in the Chip Planner



To create a logic element resource for Stratix, Cyclone, and MAX device families, perform the following steps:

1. Right-click on any available (unshaded) LE resource and click **Create Atom**. The **Create Logic Cell Atom** dialog box appears.
2. If applicable, in the **Partition** list, select the partition that the newly created atom should reside in. The default partition for newly created atoms is the top-level partition.
3. In the **Atom Name** box, type the atom name.
4. Click **OK**.

To create a combinational resource atoms for Cyclone II devices, perform the following steps:

1. Right-click on the left side of an available (unshaded) LE resource and click **Create Atom**. The **Create Cyclone II LUT Atom** dialog box appears.

2. If applicable, in the **Partition** list, select the partition that the newly created atom should reside in. The default partition for newly created atoms is the top-level partition.

3. In the **Atom Name** box, type the atom name.

4. Click **OK**.

To create a register resource atom for Cyclone II devices, perform the following steps:

1. Right-click on right side of an available (unshaded) LE resource and click **Create Atom**. The **Create Cyclone II Register Atom** dialog box appears.

2. If applicable, in the **Partition** list, select the partition that the newly created atom should reside in. The default partition for newly created atoms is the top-level partition.

3. In the **Atom Name** box, type the atom name.

4. Click **OK**.

Deleting Atoms

To delete a resource atoms, right-click on the desired resource atom in the Chip Planner and click **Delete Atom**.

You can delete a resource only after all of its fan-out connections are removed. Protected resources, such as resources in megafunctions or IP cores, cannot be deleted.

Refer to "Resource Property Editor" on page 14–21 for more details about removing fan-out connections.

Moving Atoms

You can move resource atoms by clicking on the desired resource and dragging the selected atom to a free resource atom. Moving nodes as an ECO can only be done in the ECO editing mode. Changes made while in Assignment mode create location constraints on the design and require a recompilation to incorporate the change.

Resource atoms from protected resources, such as resources of Megafunction IP cores, cannot be moved.

Check and Save Netlist changes

After making all ECOs, you can run the Fitter to incorporate the changes by clicking on the **Check and Save All Netlist Changes** icon in the Chip Planner toolbar. The Fitter compiles the ECO changes, performs design rule checks on the design, and generates a programming file.

Resource Property Editor

You can view and edit the following resources with the Resource Property Editor:

- "Logic Element" (Stratix, Stratix GX, Cyclone III, Cyclone II, Cyclone, and MAX II) on page 14–21
- "Adaptive Logic Module"(Arria GX, Stratix II, Stratix II GX, and Stratix III) on page 14–25
- "FPGA I/O Elements"(I/O resources) on page 14–27
- "PLL Properties"on page 14–43

You can view (but not edit) the following resources with the Resource Property Editor:

- RAM
- DSP blocks

Logic Element

An Altera LE contains a four-input LUT, which is a function generator that can implement any function of four variables. In addition, each LE contains a register that is fed by the output of the LUT or by an independent function generated in another LE.

You can use the Resource Property Editor to view and edit any LE in the FPGA. Open the Resource Property Editor for an LE by pointing to Locate and clicking **Locate in Resource Property Editor** in one of the following views:

- Timing Closure Floorplan
- RTL Viewer
- Technology Map Viewer
- Node Finder
- Chip Planner



For more information about LE architecture for a particular device family, refer to the device family handbook or data sheet.

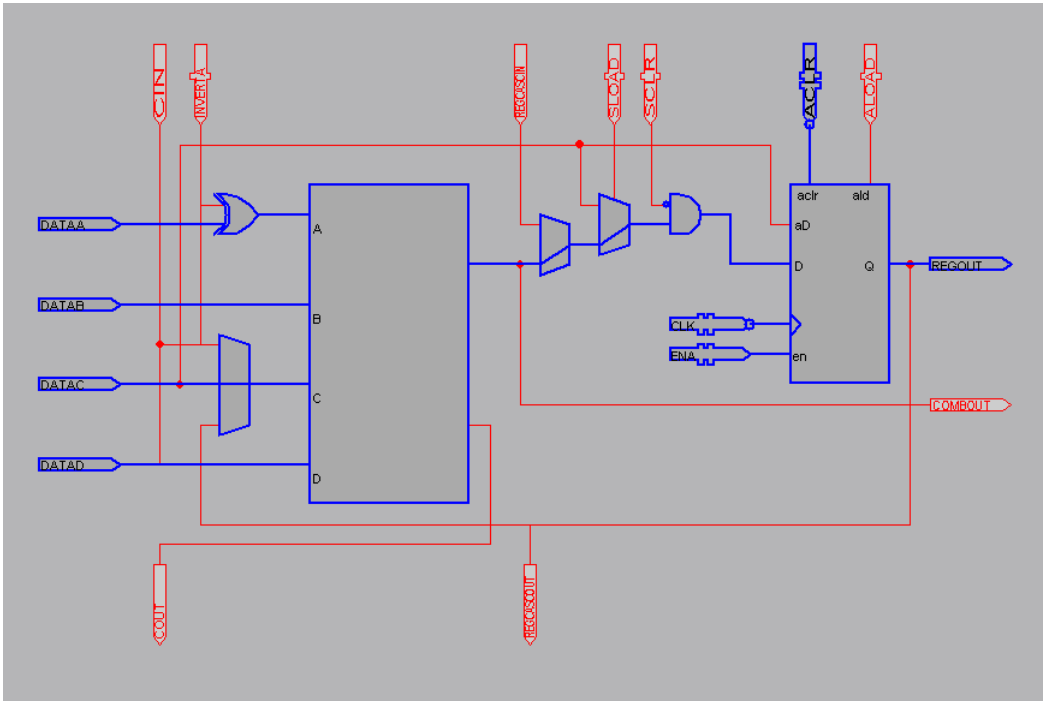
You can use the Resource Property Editor to change the following LE properties:

- Data input to the LUT
- LUT mask or LUT equation

Logic Element Schematic View

Figure 14–13 shows how the LE appears in the Resource Property Editor.

Figure 14–13. Stratix LE Architecture *Notes (1), (2)*



Notes to Figure 14–13:

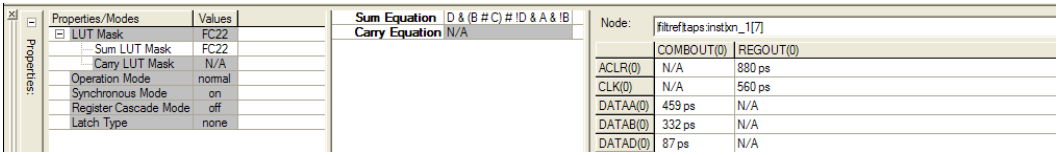
(1) By default, the Quartus II software displays the used resources in blue and the unused in gray. For Figure 14–13, the used resources are in blue and the unused resources are in red.

(2) For more information about the Stratix device's LE Architecture, refer to the *Stratix Device Handbook*.

LE Properties

Figure 14–14 shows an example of the properties that can be viewed for a selected LE in the Resource Property Editor. To view LE properties, on the View menu, click **View Properties** to view these properties.

Figure 14–14. LE Properties



Modes of Operation

LUTS in an LE can operate in either normal or arithmetic mode.

For more information about LE modes of operation, refer to volume 1 of the appropriate device handbook.

When an LE is configured in normal mode, the LUT in the LE can implement a function of four inputs.

When the LE is configured in arithmetic mode, the LUT in the LE is divided into two 3-input LUTs. The first LUT generates the signal that drives the output of the LUT, while the second LUT generates the carry-out signal. The carry-out signal can only drive a carry-in signal of another LE.

Sum and Carry Equations

You can change the logic function implemented by the LUT by changing the sum and carry equations. When the LE is configured in normal mode, you can only change the SUM equation. When the LE is configured in arithmetic mode, you can change both the SUM and the CARRY equations.

The LUT mask is the hexadecimal representation of the LUT equation output. When you change the LUT equation, the Quartus II software automatically changes the LUT mask. Conversely, when you change the LUT mask, the Quartus II software automatically computes the LUT equation.

sload and sclear Signals

Each LE register contains a synchronous load (`sload`) signal and a synchronous clear (`sclear`) signal. You can invert either the `sload` or `sclr` signal feeding into the LE. If the design uses the `sload` signal in an LE, the signal and its inversion state must be the same for all other LEs in the same LAB. For example, if two LEs in a LAB have the `sload` signal connected, both LEs must have the `sload` signal set to the same value. This is also true for the `sclr` signal.

Register Cascade Mode

When register cascade mode is enabled, the cascade-in port feeds the input to the register. The register cascade mode is used most often when the design implements shift registers. You can change the register cascade mode by connecting (or disconnecting) the cascade in port. However, if you create this port, you must ensure that the source port LE is directly above the destination LE.

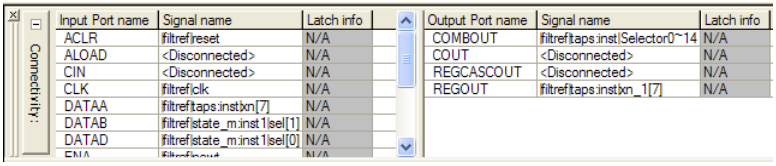
Cell Delay Table

The cell delay table describes the propagation delay from all inputs to all outputs for the selected LE.

LE Connections

On the View menu, click **View Port Connections** to view the connections that feed in and out of an LE. Figure 14–15 shows the LE connections in the Connectivity window.

Figure 14–15. View Connections



Delete an LE

To delete an LE, perform the following steps:

1. Right click the desired LE in the Resource Property Editor, point to Locate and click **Locate in Resource Property Editor**.

2. You must remove all fan-out connections from an LE prior to deletion. To delete fan-out connections, right-click each connected output signal, point to remove, and click **Fanouts**. Select all of the fan-out signals on the dialog box that appear and click **OK**.

3. Remove fan-out connections. After you locate the LE in the Resource Property Editor, delete the fan-out connections. On the right-click menu, point to Remove and click **Fanouts** on all outputs.

4. To delete an atom after all fan-out connections are removed, reference the atom back in the Chip Planner. Right-click and choose **Delete Atom**.

Adaptive Logic Module

The basic building block of logic in the Arria GX, Stratix II, Stratix II GX, and Stratix III architectures is the ALM. The ALM provides advanced features that have efficient logic utilization. Each ALM contains LUT-based resources that can be divided between two adaptive LUTs (ALUTs). With up to eight inputs to the two ALUTs, each ALM can implement various combinations of two functions. This adaptability allows the ALM to be completely backward-compatible with four-input LUT architectures. One ALM can implement any function with up to six inputs and certain seven-input functions. In addition to the adaptive LUT-based resources, each ALM contains two programmable registers, two dedicated full adders, a carry chain, a shared arithmetic chain, and a register chain. The ALM can efficiently implement various arithmetic functions and shift registers using these dedicated resources.

You can implement the following types of functions in a single ALM:

- Two independent 4-input functions
- An independent 5-input function and an independent 3-input function
- A 5-input function and a 4-input function, if they share one input
- Two 5-input functions, if they share two inputs
- An independent 6-input functions
- Two 6-input functions, if they share four inputs and share function
- Certain 7-input functions

You can use the Resource Property Editor to change the following LE properties:

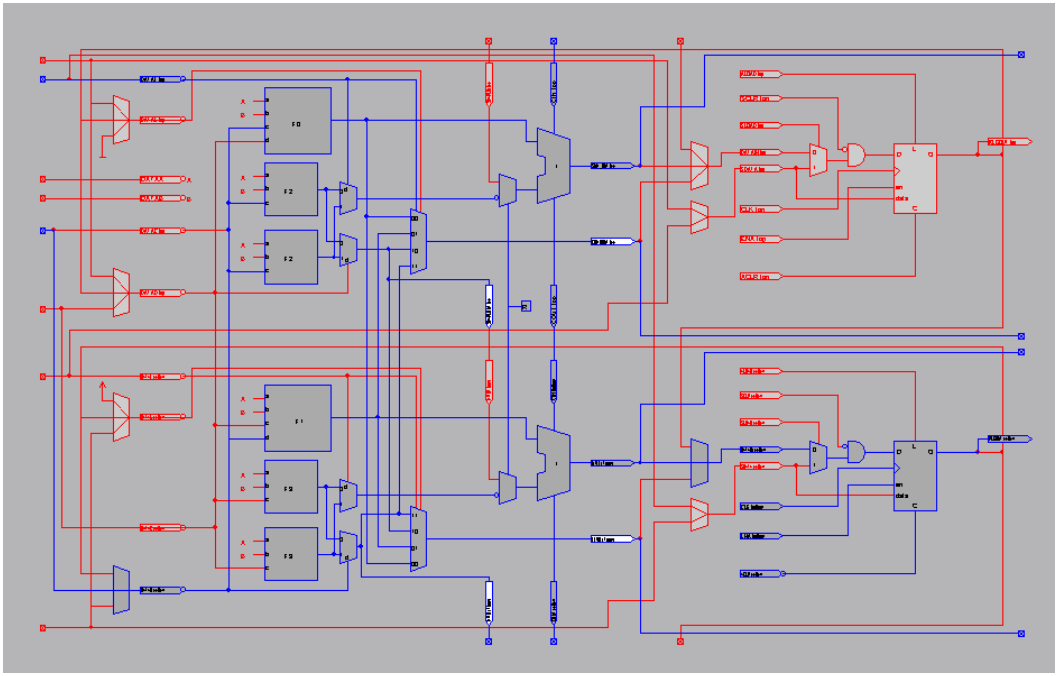
- Existing ALM atom location
- Data input to the LUT
- LUT mask or LUT equation

ALM Schematic

You can view and edit any ALM atom with the Resource Property Editor by right-clicking the ALM in the Timing Closure Floorplan, the RTL Viewer, the Node Finder, or the Chip Planner, and clicking **Locate in Resource Property Editor** (Figure 14–16).

For a detailed description of the ALM, refer to the device handbooks for Arria GX, Stratix II, Stratix II GX, or Stratix III device families.

Figure 14–16. ALM Schematic *Note (1)*



Note to Figure 14–16:

(1) By default, the Quartus II software displays the used resources in blue and the unused in gray. For Figure 14–16, the used resources are in blue and the unused resources are in red.

ALM Properties

The properties that you can display for the ALM include an equations table that shows the name and location of each of the two combinational nodes and two register nodes in the ALM, the individual LUT equations for each of the combinational nodes and the `combout`, `sumout`, `carryout`, and `shareout` equations for each combinational node.

ALM Connections

On the View menu, click **View Port Connections** to view the connections that feed in and out of an ALM.

FPGA I/O Elements

Altera FPGAs, with high-performance I/O elements including up to six registers, are equipped with support for a number of I/O standards allowing you to run your design at peak speeds.

For a detailed description of the device I/O elements, refer to the applicable device handbook.

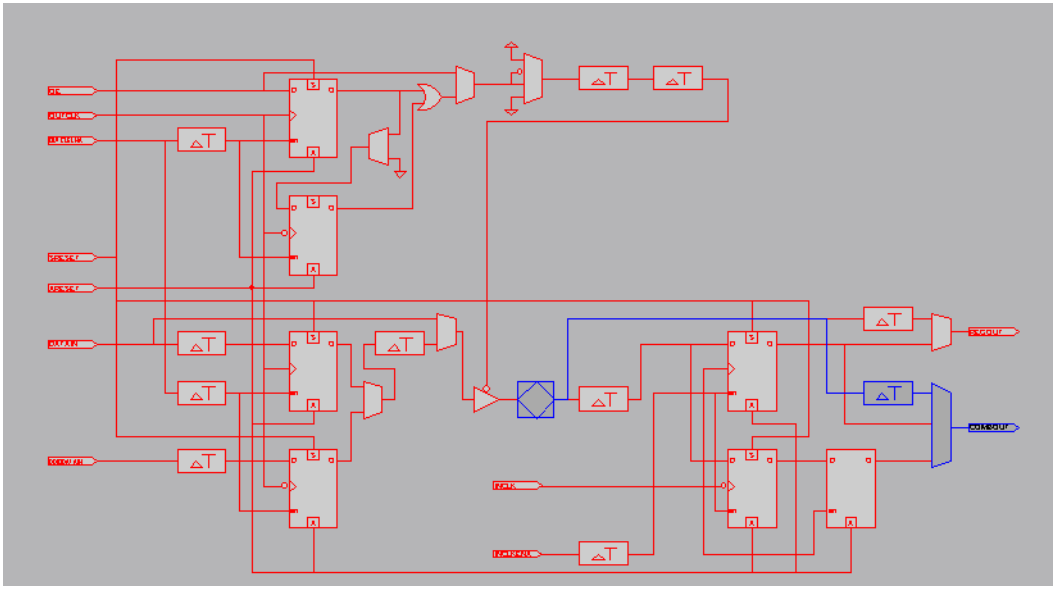
You can change the following I/O properties:

- Delay chain
- Bus hold
- Weak pull up
- Slow slew rate
- I/O standard
- Current strength
- Extend OE disable
- PCI I/O
- Register reset mode
- Register synchronous reset mode
- Register power up
- Register mode

Arria GX, Stratix II, Stratix, and Stratix GX I/O Elements

The I/O elements in Stratix series device families and Arria GX devices contain a bidirectional I/O buffer, six registers, and a latch for a complete bidirectional single data rate or DDR transfer. Figure 14–17 shows the Stratix and Stratix GX I/O element structure. The I/O element structure contains two input registers (plus a latch), two output registers, and two output enable registers.

Figure 14–17. Stratix and Stratix GX Device I/O Element and Structure *Notes (1), (2)*



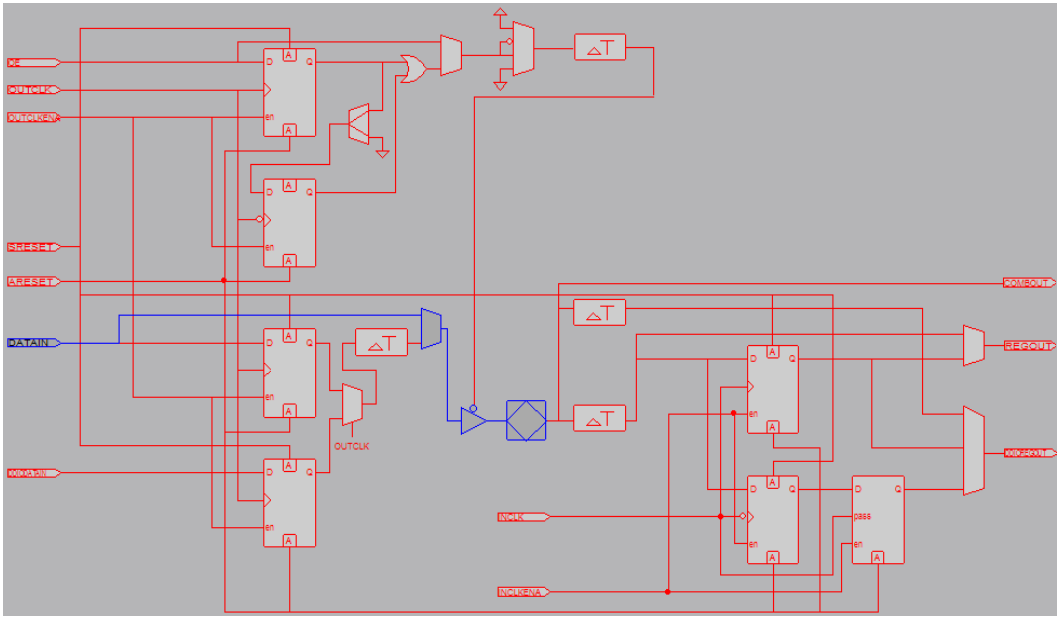
Notes to Figure 14–17:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 14–17, the used resources are in blue and the unused resources are in red.

(2) For more information about I/O elements in Stratix and Stratix GX devices, refer to the *Stratix Device Handbook* and the *Stratix GX Device Handbook*.

Figure 14–18 shows the Arria GX I/O element structures.

Figure 14–18. Arria GX Device I/O Element and Structure *Notes (1), (2), (3), (4)*



Notes to Figure 14–18:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 14–18, the used resources are in blue and the unused resources are in red.

(2) For more information about I/O elements in Stratix II devices, refer to the *Stratix II Device Handbook*.

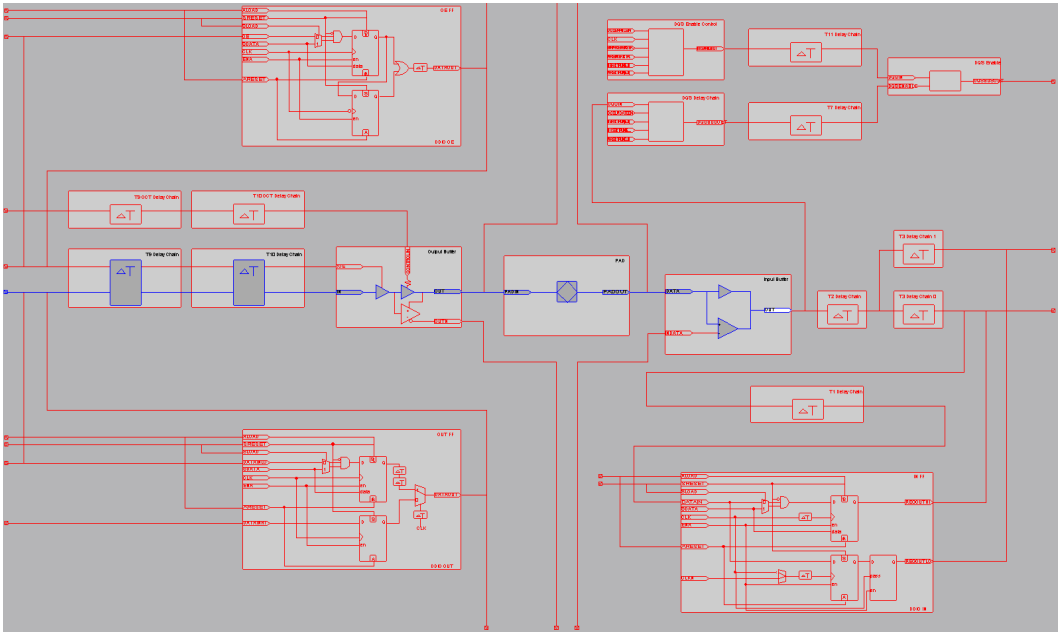
(3) This diagram also applies to Stratix II devices.

(4) Current IOE shown in a DQS pin. Non-DQS pins do no contain DQS delay circuitry.

Stratix III I/O Elements

The I/O element (IOE) in Stratix III devices contain a bi-directional I/O buffer and I/O registers to support a complete embedded bi-directional single data rate or DDR transfer (shown in Figure 14–19). The IOEs are located in I/O blocks around the periphery of the Stratix III device. The I/O registers are composed of the input path for handling data from the pin to the core, the output path for handling data from the core to the pin, and the output enable (OE) path for handling the OE signal for the output buffer. The output and OE paths are divided into output/OE registers, alignment registers, and HDR blocks. You can bypass each block of the output and output-enable path.

Figure 14–19. Stratix III Device I/O Element and Structure *Notes (1), (2)*



Notes to Figure 14–19:

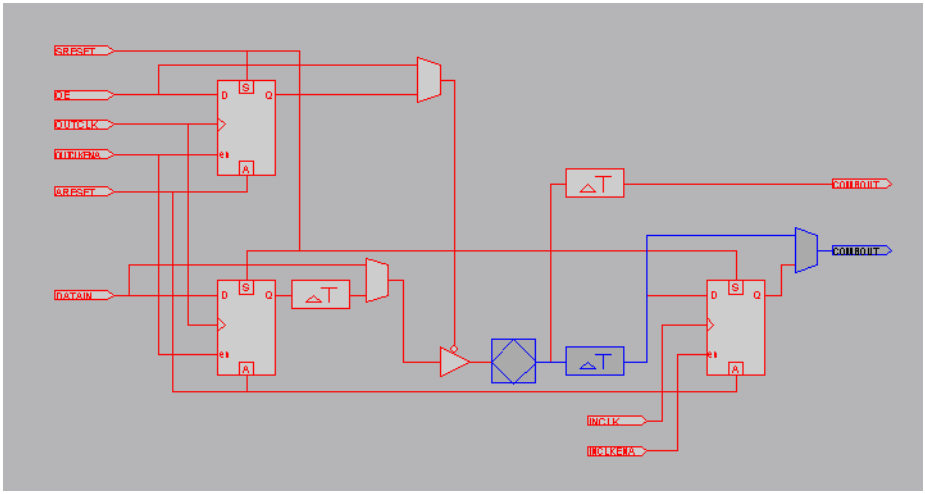
(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 14–19, the used resources are in blue and the unused resources are in red.

(2) For more information about I/O elements in Stratix III devices, refer to the *Stratix III Device Handbook*.

Cyclone II and Cyclone I/O Elements

The I/O elements in Cyclone III, Cyclone II, and Cyclone devices contain a bidirectional I/O buffer and three registers for complete bidirectional single data-rate transfer. Figure 14–20 shows the Cyclone II and Cyclone I/O element structure. The I/O element contains one input register, one output register, and one output enable register.

Figure 14–20. Cyclone II and Cyclone Device I/O Elements and Structure *Notes (1), (2)*



Notes to Figure 14–20:

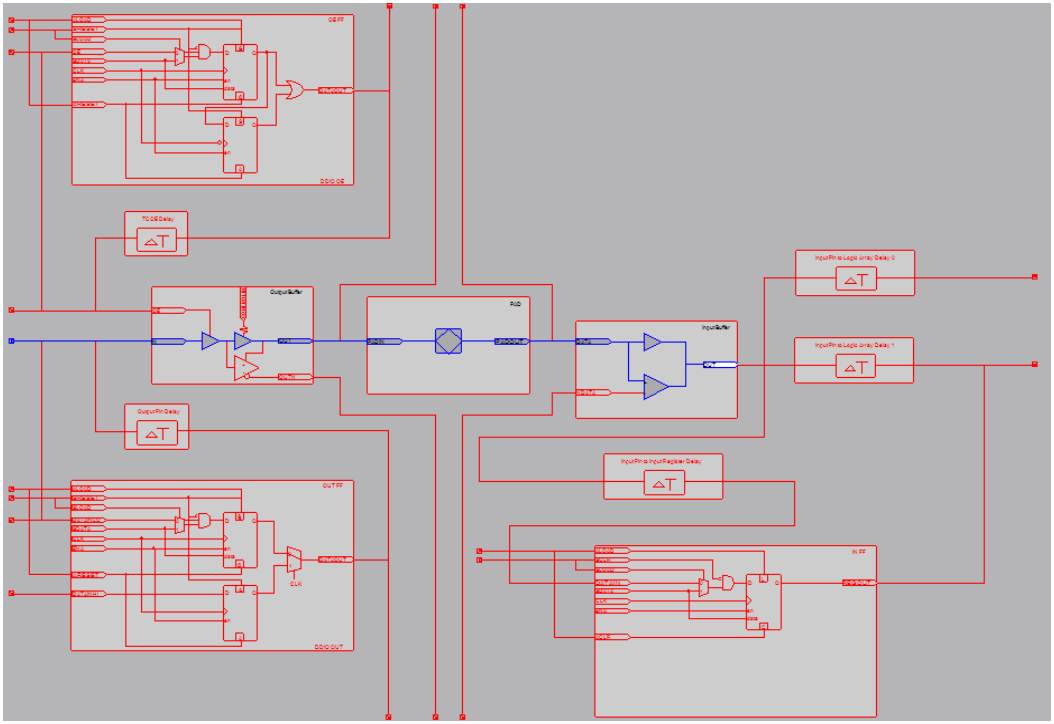
(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 14–20, the used resources are in blue and the unused resources are in red.

(2) For more information about I/O elements in Cyclone II and Cyclone devices, refer to the *Cyclone II Device Handbook* and *Cyclone Device Handbook*, respectively.

Cyclone III I/O Elements

Cyclone III device IOEs contain a bidirectional I/O buffer and five registers for complete embedded bidirectional single-data rate transfer. Figure 14–21 shows the Cyclone III IOE structure. The IOE contains one input register, two output registers, and two output-enable registers. The two output registers and two output-enable registers are utilized for double-data rate (DDR) applications. You can use the input registers for fast setup times and the output registers for fast clock-to-output times. Additionally, you can use the output-enable (OE) registers for fast clock-to-output enable timing. You can use IOEs for input, output, or bidirectional data paths.

Figure 14–21. Cyclone III Device I/O Elements and Structure (1), (2)



Notes to Figure 14–21:

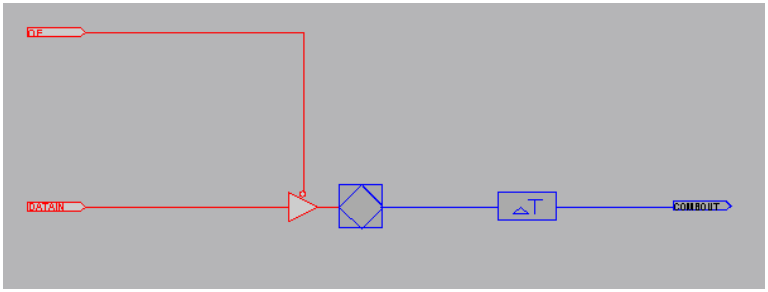
(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 14–21, the used resources are in blue and the unused resources are in red.

(2) For more information about I/O elements in Cyclone III devices, refer to the *Cyclone III Device Handbook*.

MAX II I/O Elements

MAX II device I/O elements contain a bidirectional I/O buffer. Figure 14–22 shows the MAX II I/O element structure. Registers from adjacent LABs can drive to or be driven from the I/O element's bidirectional I/O buffers.

Figure 14–22. MAX II Device I/O Elements and Structure *Notes (1), (2)*



Notes to Figure 14–22:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 14–22, the used resources are in blue and the unused resources are in red.

(2) For more information about I/O elements in MAX II devices, refer to the *MAX II Device Handbook*.

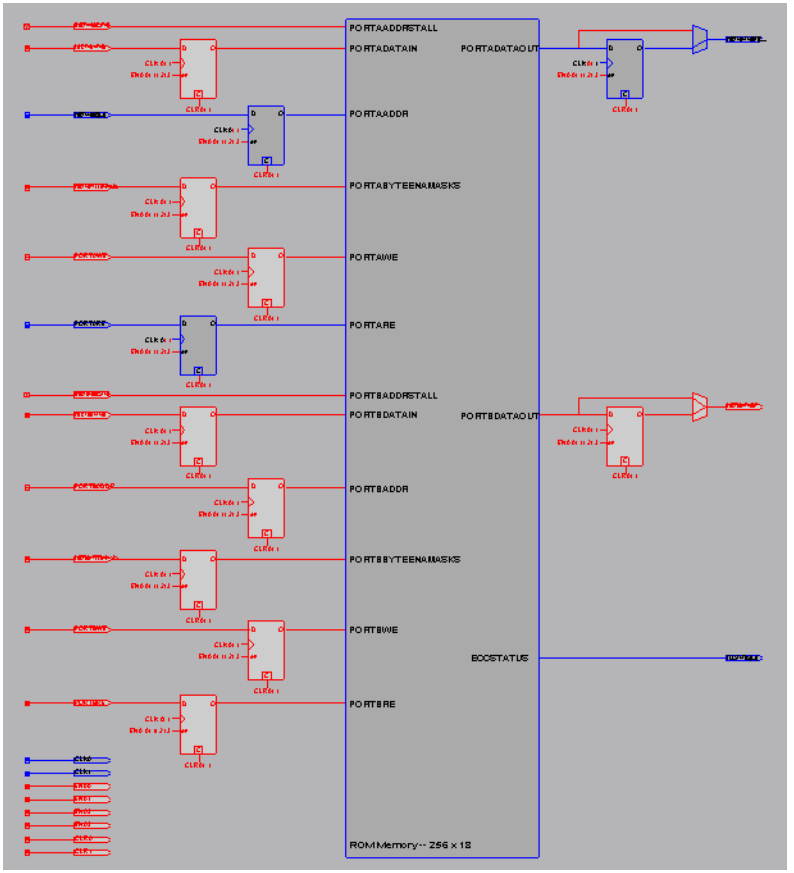
I/O Element Features in the Resource Property Editor

You use the Resource Property Editor to view, change connectivity, and edit the properties of the I/O elements. You can use the Chip Planner to change placement, delete, and create new I/O elements. You can perform all these operations on Arria GX, Stratix III, Stratix II, Stratix II GX, Stratix, Stratix GX, Cyclone III, Cyclone II, Cyclone, and MAX II devices.

FPGA RAM Blocks

In the Quartus II software beginning with version 6.1, you can view the architecture of different RAM blocks in the device. Figure 14–23 shows a M9K RAM view in a Stratix III device. You can only view the connections and properties of RAM blocks in the Resource Property Editor.

Figure 14–23. M9k RAM View in a Stratix III Device *Note (1)*



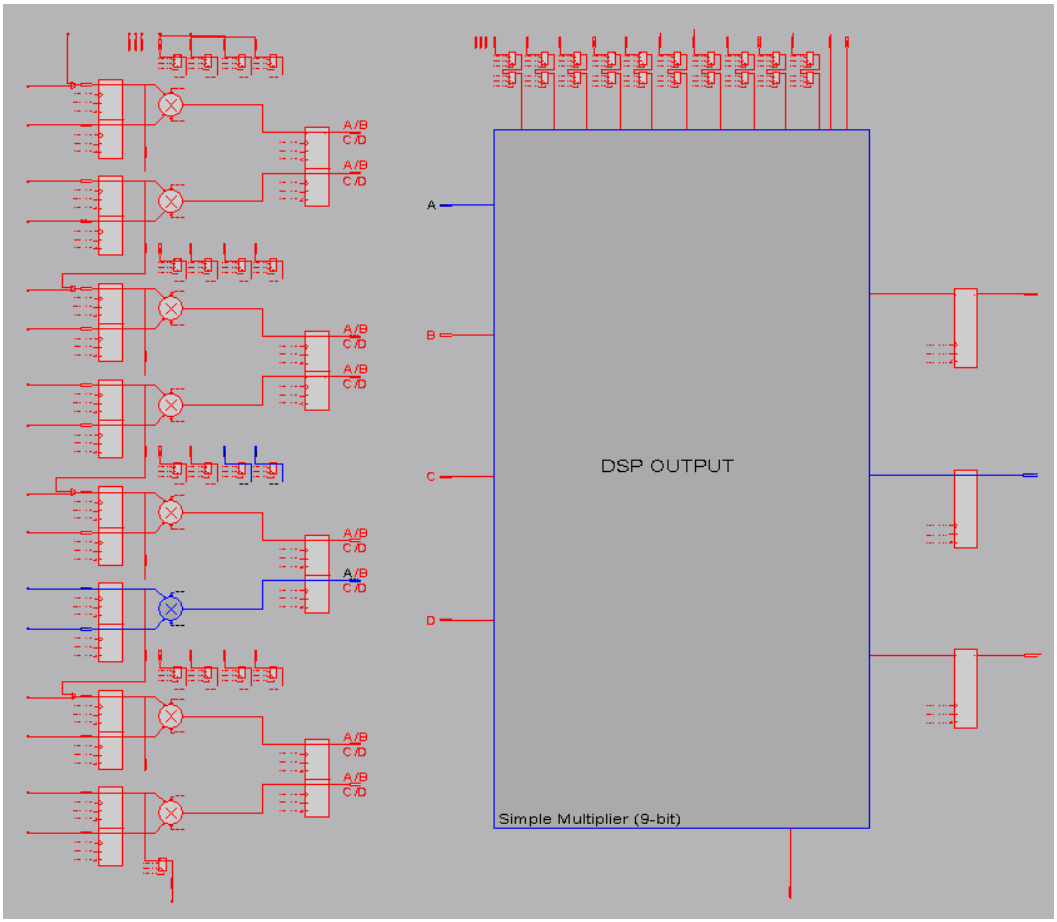
Note to Figure 14–23:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 14–23, the used resources are in blue and the unused resources are in red.

FPGA DSP Blocks

Dedicated hardware DSP circuit blocks in Altera devices provide performance benefits for the critical DSP functions in your design. Beginning with version 6.1 of the Quartus II software, you can view the architecture of DSP blocks in the Resource Property Editor for Stratix and Cyclone series of devices. Figure 14–24 shows a view of a DSP architecture in a Stratix III device. You can view the properties and connections of DSP blocks in the Resource Property Editor but you cannot edit them.

Figure 14–24. DSP Block View in a Stratix III Device *Note (1)*



Note to Figure 14–24:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 14–24, the used resources are in blue and the unused resources are in red.

Change Manager

The Change Manager maintains a record of every change that you perform with the Resource Property Editor. Each row in the Change Manager represents one ECO performed. The changes are numbered sequentially, such that the larger the number, the more recent the change.

More complex changes are marked in the Change Manager with a plus icon. You can expand a complex entry in the Change Manager by clicking the plus icon to reveal all the changes that occurred. An example of a complex change is the creation or deletion of an atom.

Table 14–1 summarizes the information shown by the Change Manager.

Table 14–1. Change Manager Information

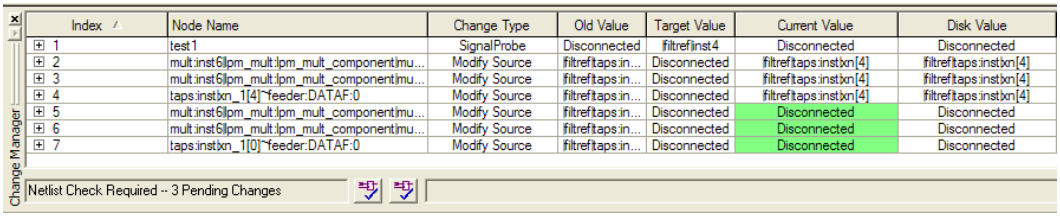
Column Name	Description
Index	Identifies, by a sequential number, change records corresponding to changes made in the Chip Planner or Resource Property Editor. In the case of complex change records, the index column identifies not only the main change, but also any component changes.
Node Name	Uniquely identifies the resource to which a change has been made.
Change Type	Identifies the type of change that has been made to the resource.
Old Value	Lists the value of the resource immediately prior to the change being made.
Target Value	Lists the desired target value (new value) that you have established using the Resource Property Editor, Chip Planner, or SignalProbe.
Current Value	Lists the value of the resource in the netlist that is currently active in memory (as opposed to the value in the netlist saved on disk, which may be different if you have made changes and not yet used the **Check & Save All Netlist Changes** command).
Disk Value	Lists the current value of the resource on disk.
Comment	Lets you add a comment to a change record in the Change Manager. To add a comment to a change record, double-click in the **Comment** field of the record you want to annotate, and type the desired comment.

After you complete all of your design modifications, check the integrity of the netlist by right-clicking in the Change Manager and clicking **Check & Save All Netlist Changes**. If the applied changes successfully pass the netlist check, they are written to disk. If the changes do not pass the netlist check, all changes made since the previous successful netlist check are reversed. Figure 14–25 shows the Change Manager.

Colored indicators in the **Current Value** and **Disk Value** columns indicate the present status of the data in those columns. Green in the **Current Value** column indicates that the change has occurred. Blue in the

Disk Value column indicates that the change has successfully passed a **Check & Save Netlist Changes** operation. Choose **Check & Save All Netlist Changes**.

Figure 14–25. Change Manager Results



For more information about SignalProbe pins, refer to the *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*.

Each line in the Change Manager represents a change record. Simple changes appear as a single line. More complex changes, which require that several actions be performed to achieve the change, appear as a single line marked by a plus icon. Click the plus icon to show all the component actions performed as part of the change.

Complex Changes in the Change Manager

Certain types of changes that you make in the Resource Property Editor or the Chip Planner (including creating or deleting atoms and changing connectivity) may appear to be self-contained, but these changes are actually composed of multiple actions. Complex changes are indicated with a plus icon in the **Index** column.

The change record in the **Change Manager** is a single-line representation of the actual change actions that occurred. You expand the change record to show the component actions that make up the change by clicking the plus icon.

After expanding a change entry in the Change Manager, you can see that creation of an atom consists of three actions:

- The creation of a new logic cell
- The creation of an output port on the newly created logic cell
- The assignment of a location index to the newly created logic cell

You cannot select individual components of a complex change record; if you select any part of a complex change record, the entire complex change record is selected.



For examples of managing changes with the Change Manager, refer to "Example of Managing Changes With the Change Manager" in the Quartus II Help.

Managing SignalProbe Signals

The SignalProbe pins that you create from the **SignalProbe Pins** dialog box are recorded in the Change Manager. After you have created a SignalProbe assignment, you can use the Change Manager to quickly disable SignalProbe assignments by selecting **Revert to Last Saved Netlist** from the right-click menu in the Change Manager.



For more information about SignalProbe pins, refer to the *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*.

Exporting Changes

You can export all your changes to a tool command language (Tcl) script, a Comma Separated Value (**.csv)** file, or a Text **(.txt)** file. The Tcl file enables you to write a script that reapplies changes that were deleted by compilation. You can also write a script that applies to other Quartus II software projects that you create. The Comma-Separated Value or Text files provide a list of changes in a tabular format. To export changes, perform the following steps:

1. On the right-click menu, click **Export Changes**.
2. Specify the Tcl file name.
3. Click **OK**.

The resulting Tcl script can also implement similar changes to another Quartus II design.

Using Incremental Compilation in the ECO Flow

Beginning with the Quartus II software version 6.1, the incremental compilation feature is turned on by default. The top-level design is automatically set to a design partition when the incremental compilation feature is on. A design partition during incremental compilation can have different netlist types (netlist types can be set to source HDL, post synthesis, or post-fit). The netlist type indicates whether that partition should be resynthesized or refit during the recompilation. Incremental

compilation saves you time and preserves the placement of unchanged partitions in your design if small changes must be made to some partitions late in the design cycle.



For more information about partitions, their netlist types and the Quartus II incremental compilation, refer to the *Quartus Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the Quartus II *Handbook*.

The behavior of ECOs during an incremental compilation depends on the netlist type of your design partitions. The Quartus II software preserves ECOs if the partition containing the ECO satisfies the following two conditions:

- The netlist type of the affected partition is set to post-fit with the Fitter preservation level set to Placement and Routing.
- There are no source code changes in the affected partition that would cause the partition to be resynthesized during recompilation.

If you have ECOs that affect multiple partitions in your design, the Quartus II software preserves your ECOs during recompilation if any of the affected partitions are set to post-fit. Note that whenever an ECO affects multiple partitions, all of the affected partitions become linked. All of the higher-level "parent" partitions up to their nearest common parent are also linked. In such cases, the connection between the partitions is actually defined outside of the two partitions immediately affected, so all the partitions must be compiled together. The linked partition inherits the netlist type of the partition with the highest level of preservation. For example, if an ECO is performed on a lower-level partition of a post-fit type and a top-level partition of a post-synthesis type, the two partitions will be linked and will have a post-fit netlist type.

If the partitions are set to use the source code or a post-synthesis netlist, the software issues a warning and the post-fit ECO changes are not included in the new compilation.

For example, if your top-level partition netlist type is set to post–synthesis, and either you have no other lower-level partitions or the lower-level partitions netlist type is also set to post-synthesis, during recompilation, your ECOs are not preserved and a warning message appears in the messages window, indicating that ECO modifications are discarded; however, all of the ECO information is retained in the Change Manager. In this case, you can apply ECOs from the change manager and perform the **Check & Save All Netlist Changes** step as described in ""ECO Flow with No Quartus II Incremental Compilation" on page 14–40.

ECO Flow with No Quartus II Incremental Compilation

If you do not use the Quartus II incremental compilation feature and have implemented ECOs, those ECOs are not preserved during recompilation of your design; however, all of the ECOs remain in the Change Manager. To apply an ECO, right click the Change Manager and click **Apply Selected Change**. (If the Change Manager window is not visible at the bottom of your screen, from the View menu, point to Utility Windows and click **Change Manager**.)

After applying the selected ECO, perform one of the following steps:

- ✓ From the menu within the Change Manager, click **Check & Save All Netlist Changes**.

or

- ✓ From Processing menu, point to Start and click **Start Check & Save All Netlist Changes**.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. The Tcl commands for controlling the Chip Planner are located in the `chip_planner` package of the `quartus_cdb` executable. A comprehensive list of Tcl commands for the Chip Planner can be found in the *Quartus Scripting Reference Manual*.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Reference Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Common ECO Applications

This section provides examples of the situations in which you might use an ECO to make a post-compilation change to your design. To help build your system quickly, you can use Chip Planner functions to perform the following activities:

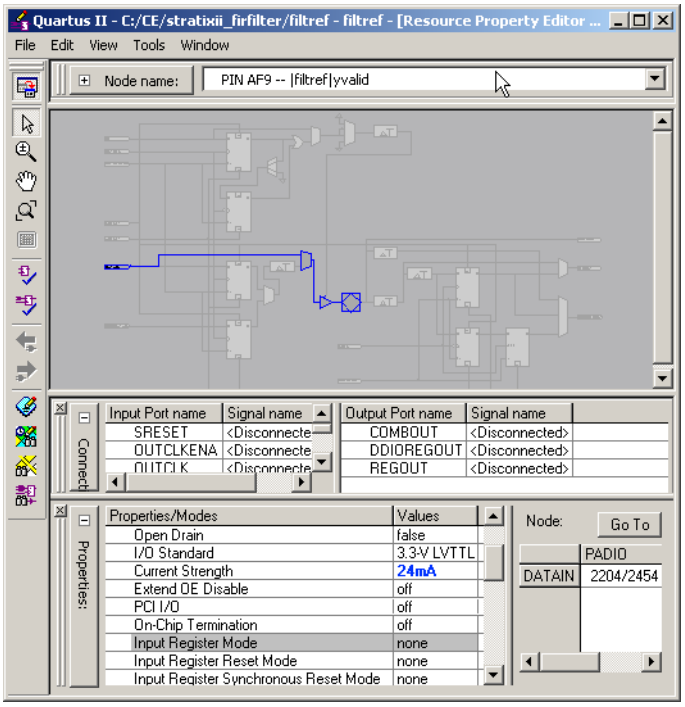
- Adjust the drive strength of an I/O using the Chip Planner
- Modifying the PLL properties using the Chip Planner (see page 14–42)

Adjust the Drive Strength of an I/O Using the Chip Planner

To adjust the drive strength of an I/O, follow the steps in this section to run the Fitter and assembler to incorporate the ECO changes into the netlist of the design.

1. In Chip Planner, select the **Post Compilation Editing (ECO)** task.
2. Locate the I/O in the **Resource Property Editor**, as shown in Figure 14–26.

Figure 14–26. I/O in the Resource Property Editor



3. Click the **Current Strength** box for the selected I/O, then click **Edit**.
4. Change the value for the desired current strength.
5. Right-click the ECO change in the Change Manager tool and click **Check & Save All Netlist Changes** to apply the ECO change.



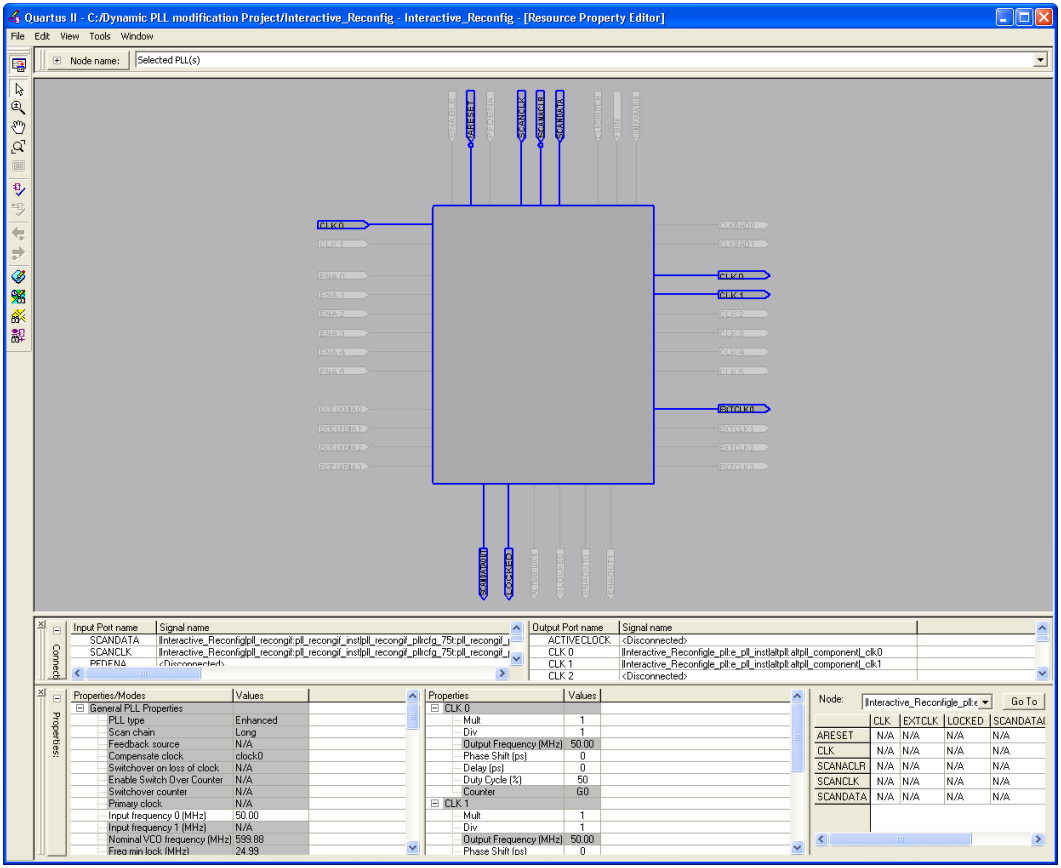
Changing the pin locations of input/output ports can be done using the ECOs flow as well. You can drag and move the signal from an existing pin location to a new location while in the Post Compilation Editing (ECO) task in the Chip Planner. Afterward, you can set **Check & Save All Netlist Changes** to compile the ECO.

Modifying the PLL Properties Using the Chip Planner

PLLs are used to modify and generate clock signals to meet design requirements. Additionally, PLLs are used for distributing clock signals to different devices in a design, reducing clock skew between devices, improving I/O timing, and generating internal clock signals.

The Resource Property Editor enables you to view and modify PLL properties to meet your design requirements. Using the Stratix PLL as an example, the rest of this section describes the adjustable PLL properties and the equations as a function of the adjustable PLL properties that govern the PLL output parameters. Figure 14–27 shows a Stratix PLL as shown in the Resource Property Editor.

Figure 14–27. The PLL View in a Stratix Device



PLL Properties

The Resource Property Editor enables you to modify PLL options, such as phase shift, output clock frequency, and duty cycle. You can also change the following PLL properties with the Resource Property Editor:

- Input frequency
- M VCO tap
- M initial
- M value
- N value
- M counter delay
- N counter delay

- M2 value
- N2 value
- SS counter
- Charge pump current
- Loop filter resistance
- Loop filter capacitance
- Counter delay
- Counter high
- Counter low
- Counter mode
- Initial
- VCO tap

Adjusting the Duty Cycle

Use the following equation to adjust the duty cycle of individual output clocks:

(1)

High % = Counter High/(Counter High + Counter Low)
Low % = Counter Low/(Counter High + Counter Low)

Adjusting the Phase Shift

Use the following equation to adjust the phase shift of an output clock of a PLL:

(2)

Phase Shift = (Period V_{CO} × 0.125 × Tap V_{CO}) + (Initial V_{CO} × Period V_{CO})

For normal mode Period V_{CO}, Tap V_{CO}, and Initial V_{CO} are governed by the following settings:

Tap V_{CO} = Counter Delay — M Tap V_{CO}
Initial V_{CO} = Counter Initial — M Initial
Period V_{CO} = In Clock Period × N / M

For external feedback mode, Tap V_{CO}, Initial V_{CO}, and Period V_{CO} are governed by the following settings:

Tap V_{CO} = Counter Delay — M Tap V_{CO}
Initial V_{CO} = Counter Initial — M Initial
Period V_{CO} = In Clock Period × N / (M + Counter High + Counter Low)



For a detailed description of the settings, refer to the Quartus II Help. For more information about Stratix device PLLs, refer to the *Stratix Architecture* chapter in volume 1 of the *Stratix Device Handbook*. For more information about PLLs in Arria GX, Stratix II, Cyclone II, and Cyclone devices, refer to the appropriate device handbook.

Adjusting the Output Clock Frequency

Use the following equation to adjust the PLL output clock in normal mode:

(3)

$$\text{Output Clock Frequency} = \text{Input Frequency} \cdot \frac{\text{M initial}}{\text{N initial} + \text{Counter High} + \text{Counter Low}}$$

Use the following equation to adjust the PLL output clock in external feedback mode:

(4)

$$\text{OUTCLK} = \text{INCLK} \cdot \frac{\text{M initial} + \text{External Feedback Counter High} + \text{External Feedback Counter Low}}{\text{N initial} + \text{Counter High} + \text{Counter Low}}$$

Adjusting the Spread Spectrum

Use the following equation to adjust the spread spectrum for your PLL:

(5)

$$\%\text{spread} = 1 - \frac{M_2 N_1}{M_1 N_2}$$

Post ECO Steps

This section describes the operations you can perform after making an ECO change with the Chip Planner.

Performing Static Timing Analysis

After you make an ECO change with the Chip Planner, you must perform static timing analysis of your design with either the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer to ensure that your changes have not adversely affected your design's timing performance.

For example, when you turn on one of the delay chain settings for a specific pin, you change the I/O timing. Therefore, to ensure that all timing requirements are still met, you should perform static timing analysis.

Whenever you change your design using the Chip Planner, Altera also recommends that you perform a gate-level timing simulation on your design with either the Quartus II Simulator or a third-party EDA simulation tool.



For more information about performing a static timing analysis of your design, refer to the Quartus II *Classic Timing Analyzer* or Quartus II *TimeQuest Timing Analyzer* chapters in volume 3 of the Quartus II *Handbook*.

Generating a Programming File

After you have performed simulation and static timing analysis and are confident that the changes meet your design requirements, generate a new programming file with the Quartus II Assembler. Use the programming file to implement your modified design in an Altera device.



For more information about programming your FPGA, refer to the *Quartus II Programmer* chapter in volume 3 of the *Quartus II Handbook*.

Conclusion

As the time-to-market pressure mounts, it is more and more important to produce a fully functional design in the shortest amount of time. To address this challenge, Altera developed the Chip Planner in the Quartus II software suite. The Chip Planner enables you to analyze and modify your design floorplan. Also, ECO changes made with the Chip Planner do not require a full recompilation, eliminating the lengthy process of RTL modification, resynthesis, and another place-and-route cycle. In summary, the Chip Planner shortens the verification cycle and brings timing closure to your design in a shorter period of time.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Cyclone Device Handbook*
- *MAX II Device Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Programmer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*
- *Stratix Architecture* chapter in volume 1 of the *Stratix Device Handbook*
- *Stratix Device Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 14–2 shows the revision history for this chapter.

Table 14–2. Document Revision History

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 14–47.	—
May 2007 v7.1.0	Initial Release.	—



Quartus II Version 7.2 Handbook

Volume 3: Verification



101 Innovation Drive
San Jose, CA 95134
www.altera.com

QII5V3-7.2

Copyright © 2007 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

I.S. EN ISO 9001

Contents

Chapter Revision Dates ..... **xxix**

About this Handbook ..... **xxxi**
How to Contact Altera xxxi
Third-Party Software Product Information xxxi
Typographic Conventions xxxii

Section I. Simulation

Chapter 1. Quartus II Simulator
Introduction 1–1
Simulation Flow 1–1
Functional Simulation 1–3
Timing Simulation 1–4
Timing Simulation Using Fast Timing Model Simulation 1–4
Waveform Editor 1–5
Creating VWFs 1–5
Count Value 1–11
Clock 1–11
Arbitrary Value 1–12
Random Value 1–13
Generating a Testbench 1–13
Grid Size 1–14
Time Bars 1–14
Stretch or Compress a Waveform Interval 1–15
End Time 1–16
Arrange Group or Bus in LSB or MSB Order 1–17
Simulator Settings 1–17
Simulation Verification Options 1–21
Simulation Output Files Options 1–24
Simulation Report 1–25
Simulation Waveform 1–25
Simulating Bidirectional Pin 1–26
Logical Memories Report 1–27
Simulation Coverage Reports 1–27
Comparing Two Waveforms 1–28
Debugging with the Quartus II Simulator 1–29
Breakpoints 1–29
Updating Memory Content 1–30

Last Simulation Vector Outputs 1–30
Conventional Debugging Process 1–30
Accessing Internal Signals for Simulation 1–30
Scripting Support 1–32
Conclusion 1–33
Referenced Documents 1–33
Document Revision History 1–34

Chapter 2. Mentor Graphics ModelSim Support

Introduction 2–1
Background 2–1
Software Compatibility 2–3
Altera Design Flow with ModelSim or ModelSim-Altera Software 2–3
Functional RTL Simulation 2–5
Functional Simulation Libraries 2–5
lpm Simulation Models 2–5
Altera Megafunction Simulation Models 2–6
Low-Level Primitive Simulation Models 2–7
Simulating VHDL Designs 2–7
Create Simulation Libraries 2–7
Create Simulation Libraries Using the ModelSim GUI 2–8
Create Simulation Libraries Using the ModelSim Command Prompt 2–8
Compile Simulation Models into Simulation Libraries 2–8
Compile Simulation Models into Simulation Libraries Using the ModelSim GUI 2–8
Compile Simulation Models into Simulation Libraries at the ModelSim Command Prompt 2–9
Compile Testbench and Design Files into Work Library 2–9
Compile Testbench and Design Files into Work Library Using the ModelSim Command Prompt 2–9
Loading the Design 2–9
Loading the Design Using the ModelSim Command Prompt 2–10
Running the Simulation 2–10
Running the Simulation Using the ModelSim Command Prompt 2–10
Simulating Verilog HDL Designs 2–10
Create Simulation Libraries 2–10
Create Simulation Libraries Using the ModelSim GUI 2–11
Create Simulation Libraries Using the ModelSim Command Prompt 2–11
Compile Simulation Models into Simulation Libraries 2–11
Compile Simulation Models into Simulation Libraries Using the ModelSim GUI 2–11
Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt 2–12
Compile Testbench and Design Files into Work Library 2–12
Compile Testbench and Design Files into Work Library Using the ModelSim Command Prompt 2–12

Loading the Design 2–12
Loading a Design Using the ModelSim Command Prompt 2–13
Running the Simulation 2–13
Running the Simulation Using the ModelSim Command Prompt 2–13
Verilog HDL Functional RTL Simulation with Altera Memory Blocks 2–13
Post-Synthesis Simulation 2–16
Generating a Post-Synthesis Simulation Netlist 2–16
Simulating VHDL Designs 2–17
Create Simulation Libraries 2–17
Create Simulation Libraries Using the ModelSim GUI 2–17
Create Simulation Libraries Using the ModelSim Command Prompt 2–18
Compile Simulation Models into Simulation Libraries Using the ModelSim GUI ... 2–18
Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt 2–18
Compile Testbench and VHDL Output File into Work Library 2–18
Compile Testbench and VHDL Output File into Work Library Using ModelSim Command Prompt 2–19
Loading the Design 2–19
Loading the Design Using the ModelSim Command Prompt 2–19
Running the Simulation 2–19
Running the Simulation Using the ModelSim Command Prompt 2–20
Simulating Verilog HDL Designs 2–20
Create Simulation Libraries 2–20
Create Simulation Libraries Using the ModelSim GUI 2–20
Create Simulation Libraries Using the ModelSim Command Prompt 2–20
Compile Simulation Models into Simulation Libraries Using the ModelSim GUI ... 2–21
Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt 2–21
Compile Testbench and Verilog Output File into Work Library 2–21
Compile Testbench and Verilog Output File into Work Library Using the ModelSim Command Prompt 2–21
Loading the Design 2–22
Loading the Design Using the ModelSim Command Prompt 2–22
Running the Simulation 2–22
Running the Simulation Using the ModelSim Command Prompt 2–23
Gate-Level Timing Simulation 2–23
Generating a Gate-Level Timing Simulation Netlist 2–23
Generating a Different Timing Model 2–24
Operating Condition Example: Generate All Timing Models for Stratix III Devices 2–25
Perform Timing Simulation Using Post-synthesis Netlist 2–26
Gate-Level Simulation Libraries 2–27
Simulating VHDL Designs 2–29
Create Simulation Libraries 2–30
Create Simulation Libraries Using the ModelSim GUI 2–30
Create Simulation Libraries Using the ModelSim Command Prompt 2–31
Compile Simulation Models into Simulation Libraries Using the ModelSim GUI ... 2–31
Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt 2–31

Compile Testbench and VHDL Output File into Work Library 2–31
Compile Testbench and VHDL Output File into Work Library Using the ModelSim Command Prompt 2–32
Loading the Design 2–32
Loading a Design Using the ModelSim Command Prompt 2–33
Running the Simulation 2–33
Running a Simulation Using the ModelSim Command Prompt 2–33
Simulating Verilog HDL Designs 2–33
Create Simulation Libraries 2–33
Create Simulation Libraries Using the ModelSim GUI 2–34
Create Simulation Libraries Using the ModelSim Command Prompt 2–34
Compile Simulation Models into Simulation Libraries Using the ModelSim GUI ... 2–34
Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt 2–35
Compile Testbench and Verilog Output File into Work Library 2–35
Compile Testbench and Verilog Output File into Work Libraries Using the ModelSim Command Prompt 2–35
Loading the Design 2–35
Loading the Design Using the ModelSim Command Prompt 2–36
Running the Simulation 2–36
Running the Simulation Using the ModelSim Command Prompt 2–36
Simulating Designs that Include Transceivers 2–37
Stratix GX Functional Simulation 2–37
Example: Performing Functional Simulation for Stratix GX in Verilog HDL 2–37
Example: Performing Functional Simulation for Stratix GX in VHDL 2–37
Stratix GX Post-Fit (Timing) Simulation 2–38
Example: Performing Timing Simulation for Stratix GX in Verilog HDL 2–38
Example: Performing Timing Simulation for Stratix GX in VHDL 2–39
Stratix II GX Functional Simulation 2–39
Example: Performing Functional Simulation for Stratix II GX in Verilog HDL 2–40
Example: Performing Functional Simulation for Stratix II GX in VHDL 2–41
Stratix II GX Post-Fit (Timing) Simulation 2–41
Example: Performing Timing Simulation for Stratix II GX in Verilog HDL 2–42
Example: Performing Timing Simulation for Stratix II GX in VHDL 2–42
Transport Delays 2–43
+transport_path_delays 2–43
+transport_int_delays 2–43
Using the NativeLink Feature with ModelSim 2–44
Setting Up NativeLink 2–44
Performing an RTL Simulation Using NativeLink 2–44
Performing a Gate-Level Simulation Using NativeLink 2–47
Setting Up a Testbench 2–48
Creating a Testbench 2–49
Scripting Support 2–50
Generating a Post-Synthesis Simulation Netlist for ModelSim 2–50
Tcl Commands 2–50
Command Prompt 2–50
Generating a Gate-Level Timing Simulation Netlist for ModelSim 2–51
Tcl Commands 2–51

Command Line 2–51
Software Licensing and Licensing Setup 2–51
LM_LICENSE_FILE Variable 2–52
Conclusion 2–52
Referenced Documents 2–52
Document Revision History 2–53

Chapter 3. Synopsys VCS Support

Introduction 3–1
Software Requirements 3–1
Using VCS in the Quartus II Design Flow 3–3
Using VCS in the Quartus II Design Flow 3–3
Functional Simulations 3–4
Megafunctions Requiring Atom Libraries 3–5
Functional RTL Simulation with Altera Memory Blocks 3–5
Compiling Functional Library Files with Compiler Directives 3–5
Post-Synthesis Simulation 3–6
Generating a Post-Synthesis Simulation Netlist 3–6
Gate-Level Timing Simulation 3–8
Generating a Gate-Level Timing Simulation Netlist 3–8
Generating Different Timing Model 3–9
Operating Condition Example: Generate All Timing Models for Stratix III Devices 3–10
Perform Timing Simulation Using Post-Synthesis Netlist 3–11
Common VCS Software Compiler Options 3–11
Using VirSim 3–12
Debugging Support Command-Line Interface 3–12
Simulating Designs that Include Transceivers 3–13
Stratix GX Functional Simulation 3–13
Example of Compiling Library Files for Functional Stratix GX Simulation in Verilog HDL 3–13
Stratix GX Post-Fit (Timing) Simulation 3–13
Example of Compiling Library Files for Timing Stratix GX Simulation in Verilog HDL 3–14
Stratix II GX Functional Simulation 3–14
Example of Compiling Library Files for Functional Stratix II GX Simulation in Verilog HDL 3–15
Stratix II GX Post-Fit (Timing) Simulation 3–16
Example of Compiling Library Files for Timing Stratix II GX Simulation in Verilog HDL 3–16
Using PLI Routines with the VCS Software 3–16
Preparing and Linking C Programs to Verilog HDL Code 3–16
Transport Delays 3–17
+transport_path_delays 3–17
+transport_int_delays 3–17
Using NativeLink with the VCS Software 3–18
Setting Up NativeLink 3–18
Performing an RTL Simulation Using NativeLink 3–18

Performing a Gate-Level Simulation Using NativeLink 3–20
Setting Up a Testbench 3–21
Creating a Testbench 3–22
Scripting Support 3–23
Generating a Post-Synthesis Simulation Netlist for VCS 3–23
Tcl Commands 3–23
Command Prompt 3–23
Generating a Gate-Level Timing Simulation Netlist for VCS 3–23
Tcl Commands 3–24
Command Prompt 3–24
Conclusion 3–24
Referenced Documents 3–24
Document Revision History 3–25

Chapter 4. Cadence NC-Sim Support

Introduction 4–1
Software Requirements 4–1
Simulation Flow Overview 4–3
Operation Modes 4–4
Quartus II Software and NC Simulation Flow Overview 4–5
Functional and RTL Simulation 4–6
Create Libraries 4–6
Basic Library Setup 4–7
Using Multiple cds.lib Files 4–7
Create a cds.lib File in Command-Line Mode 4–8
Create a cds.lib File in GUI Mode 4–8
LPM Functions, Altera Megafunctions, and Altera Primitives Libraries 4–9
Megafunctions Requiring Atom Libraries 4–11
Simulating a Design with Memory 4–11
Compile Source Code and Testbenches 4–13
Compilation in Command-Line Mode 4–13
Compilation in GUI Mode 4–14
Elaborate Your Design 4–15
Elaboration in Command-Line Mode 4–15
Elaboration in GUI Mode 4–16
Add Signals to View 4–17
Adding Signals in Command-Line Mode 4–17
Adding Signals in GUI Mode 4–18
Simulate Your Design 4–20
Functional/RTL Simulation in Command-Line Mode 4–21
Functional/RTL Simulation in GUI Mode 4–21
Post-Synthesis Simulation 4–22
Quartus II Simulation Output Files 4–22
Create Libraries 4–23

Compile Project Files and Libraries 4–23
Elaborate Your Design 4–23
Add Signals to the View 4–23
Simulate Your Design 4–24
Gate-Level Timing Simulation 4–24
Generating a Gate-Level Timing Simulation Netlist 4–24
Generating a Different Timing Model 4–25
Operating Condition Example: Generate All Timing Models for Stratix III and Cyclone III Devices 4–26
Perform Timing Simulation Using Post-Synthesis Netlist 4–27
Quartus II Timing Simulation Libraries 4–27
Create Libraries 4–28
Compile the Project Files and Libraries 4–28
Elaborate Your Design 4–28
Compiling the Standard Delay Output File (VHDL Only) in Command-Line Mode 4–29
Compiling the Standard Delay Output File (VHDL Only) in GUI Mode 4–30
Add Signals to View 4–30
Simulate Your Design 4–31
Simulating Designs that Include Transceivers 4–31
Stratix GX Functional Simulation 4–31
Example of Compiling Library Files for Functional Stratix GX Simulation in Verilog HDL 4–31
Example of Compiling Library Files for Functional Stratix GX Simulation in VHDL 4–31
Stratix GX Post-Fit (Timing) Simulation 4–32
Example of Compiling Library Files for Timing Stratix GX Simulation in Verilog HDL 4–32
Example of Compiling Library Files for Timing Stratix GX Simulation in VHDL 4–32
Stratix II GX Functional Simulation 4–33
Example of Compiling Library Files for Functional Stratix II GX Simulation in Verilog HDL 4–34
Example of Compiling Library Files for Functional Stratix II GX Simulation in VHDL 4–35
Stratix II GX Post-Fit (Timing) Simulation 4–35
Example of Compiling Library Files for Timing Stratix II GX Simulation in Verilog HDL 4–35
Example of Compiling Library Files for Timing Stratix II GX Simulation in VHDL 4–36
Pulse Reject Delays 4–36
-PULSE_R 4–36
-PULSE_INT_R 4–36
Using the NativeLink Feature with NC-Sim 4–37
Setting Up NativeLink 4–37
Performing an RTL Simulation Using NativeLink 4–37
Performing a Gate Level Simulation Using NativeLink 4–40
Setting Up a Testbench 4–40
Creating a Testbench 4–42
Incorporating PLI Routines 4–43
Dynamically Link a PLI Library 4–43

Dynamically Load a PLI Library 4–44
Statically Link the PLI Library with NC-Sim 4–47
Scripting Support 4–48
Generate NC-Sim Simulation Output Files 4–49
Tcl Commands: 4–49
Command Prompt 4–49
Conclusion 4–49
Referenced Documents 4–50
Document Revision History 4–50

Chapter 5. Simulating Altera IP in Third-Party Simulation Tools

Introduction 5–1
IP Functional Simulation Flow 5–1
Verilog and VHDL IP Functional Simulation (IPFS) Models 5–2
Instantiate the IP in Your Design 5–3
Perform Simulation 5–4
Simulating Altera IP Using the Quartus II NativeLink Feature 5–4
Set up a Quartus II Project 5–5
Select the Third-Party Simulation Tool 5–5
Specify the Path for the Third-Party Simulator 5–7
Specify the Testbench Settings 5–7
Analyze and Elaborate the Quartus II Project 5–9
Run RTL Functional Simulation 5–9
Simulating Altera IP Without the Quartus II NativeLink Feature 5–9
Design Language Examples 5–11
Verilog HDL Example: Simulating the IPFS Model in the ModelSim Software 5–11
VHDL Example: Simulating the IPFS Model in the ModelSim Software 5–12
NC-VHDL Example: Simulating the IPFS Model in the NC-VHDL Software 5–14
Verilog HDL Example: Simulating Your IPFS Model in VCS 5–15
Single-Step Process 5–15
Two-Step Process (Compilation and Simulation) 5–16
Conclusion 5–16
Referenced Documents 5–16
Document Revision History 5–17

Section II. Timing Analysis

Chapter 6. The Quartus II TimeQuest Timing Analyzer

Introduction 6–1
Getting Started with the Quartus II TimeQuest Timing Analyzer 6–2
Setting Up the Quartus II TimeQuest Timing Analyzer 6–2
Compilation Flow with the Quartus II TimeQuest Timing Analyzer Guidelines 6–3
Running the Quartus II TimeQuest Timing Analyzer 6–4
Directly from the Quartus II Software 6–4
Stand-Alone Mode 6–5

Command-Line Mode 6–5
Timing Analysis Overview 6–7
Clock Analysis 6–11
Clock Setup Check 6–11
Clock Hold Check 6–13
Recovery and Removal 6–15
Multicycle Paths 6–16
Specify Design Timing Requirements 6–19
Create a Timing Netlist 6–19
Specify Timing Constraints 6–20
Generate SDC Constraint Reports 6–21
The Quartus II TimeQuest Timing Analyzer Flow Guidelines 6–22
Create a Timing Netlist 6–22
Read the Synopsys Design Constraints File 6–22
Update Timing Netlist 6–23
Generate Timing Reports 6–23
Collections 6–23
Application Examples 6–25
Constraints Files 6–25
Fitter and Timing Analysis SDC Files 6–26
Specifying SDC Files for Place-and-Route 6–26
Specifying SDC Files for Static Timing Analysis 6–26
Synopsys Design Constraints File Precedence 6–27
Clock Specification 6–28
Clocks 6–28
Generated Clocks 6–29
Virtual Clocks 6–32
Multi-Frequency Clocks 6–33
Automatic Clock Detection 6–34
Derive PLL Clocks 6–35
Default Clock Constraints 6–37
Clock Groups 6–37
Clock Effect Characteristics 6–39
Clock Latency 6–39
Clock Uncertainty 6–40
Derive Clock Uncertainty 6–41
Inter-Clock Transfers 6–42
Intra-Clock Transfers 6–43
I/O Interface Clock Transfers 6–43
I/O Specifications 6–45
Input and Output Delay 6–45
Set Input Delay 6–45
Set Output Delay 6–47
Timing Exceptions 6–48
Precedence 6–49
False Path 6–49
Minimum Delay 6–50

Maximum Delay 6–52
Multicycle Path 6–53
Clock-as-Data Analysis 6–55
Application Examples 6–55
Constraint and Exception Removal 6–57
Timing Reports 6–58
report_timing 6–58
report_clock_transfers 6–62
report_clocks 6–63
report_min_pulse_width 6–63
report_net_timing 6–65
report_sdc 6–66
report_ucp 6–66
report_path 6–68
report_datasheet 6–69
report_rskm 6–70
report_tccs 6–70
report_path 6–71
check_timing 6–73
report_clock_fmax_summary 6–75
create_timing_summary 6–76
Timing Analysis Features 6–77
Multi-Corner Analysis 6–77
Advanced I/O Timing and Board Trace Model Assignments 6–79
Wildcard Assignments and Collections 6–79
Resetting a Design 6–81
The TimeQuest Timing Analyzer GUI 6–82
The Quartus II Software Interface and Options 6–83
View Pane 6–85
View Pane: Splitting 6–85
View Pane: Removing Split Windows 6–86
Tasks Pane 6–87
Opening a Project and Writing a Synopsys Design Constraints File 6–87
Netlist Setup Folder 6–88
Reports Folder 6–88
Macros Folder 6–89
Console Pane 6–90
Report Pane 6–90
Constraints 6–90
Name Finder 6–92
Target Pane 6–94
SDC Editor 6–94
Conclusion 6–95
Referenced Documents 6–95
Document Revision History 6–96

Chapter 7. Switching to the Quartus II TimeQuest Timing Analyzer

Introduction 7–1
Benefits of Switching to the Quartus II TimeQuest Analyzer 7–1
Chapter Contents 7–2
Switching to the Quartus II TimeQuest Analyzer 7–2
Compile Your Design 7–2
Create an SDC File 7–3
Conversion Utility 7–3
Perform Timing Analysis with the Quartus II TimeQuest Timing Analyzer 7–4
Run the Quartus II TimeQuest Analyzer 7–4
Set the Default Timing Analyzer 7–4
Differences Between Quartus II TimeQuest and Quartus II Classic Timing Analyzers 7–5
Terminology 7–5
Netlist 7–6
Collections 7–7
Constraints 7–7
Constraint Files 7–7
Constraint Entry 7–8
Time Units 7–9
MegaCore Functions 7–9
Bus Name Format 7–10
Constraint File Priority 7–10
Constraint Priority 7–11
Ambiguous Constraints 7–12
Clocks 7–13
Related and Unrelated Clocks 7–13
Clock Offset 7–14
Clock Latency 7–15
Offset and Latency Example 7–15
Clock Offset Scenario 7–16
Clock Latency Scenario 7–17
Clock Uncertainty 7–18
Derived and Generated Clocks 7–19
Automatic Clock Detection 7–19
derive_clocks Command 7–20
derive_pll_clocks Command 7–22
Hold Relationship 7–23
Clock Objects 7–23
Hold Multicycle 7–24
Fitter Behavior 7–27
Fitter Performance 7–27
Reporting 7–27
Paths and Pairs 7–28
Default Reports 7–28
Netlist Names 7–29
Non-Integer Clock Periods 7–29
Other Features 7–30

Scripting API 7–32
Timing Assignment Conversion 7–33
Setup Relationship 7–33
Hold Relationship 7–34
Clock Latency 7–34
Clock Uncertainty 7–34
Inverted Clock 7–35
Not a Clock 7–35
Default Required f_{MAX} Assignment 7–35
Virtual Clock Reference 7–36
Clock Settings 7–37
Multicycle 7–37
Clock Enable Multicycle 7–38
I/O Constraints 7–38
Input and Output Delay 7–39
t_{SU} Requirement 7–40
t_H Requirement 7–43
t_{CO} Requirement 7–45
Minimum t_{CO} Requirement 7–48
t_{PD} Requirement 7–50
Combinational Path Delay Scenario 7–50
Minimum t_{PD} Requirement 7–51
Cut Timing Path 7–52
Maximum Delay 7–52
Minimum Delay 7–52
Maximum Clock Arrival Skew 7–53
Maximum Data Arrival Skew 7–53
Constraining Skew on an Output Bus 7–53
Conversion Utility 7–55
Unsupported Global Assignments 7–56
Recommended Global Assignments 7–56
Clock Conversion 7–58
Instance Assignment Conversion 7–59
PLL Phase Shift Conversion 7–61
t_{CO} Requirement Conversion 7–62
Entity-Specific Assignments 7–63
Paths between Unrelated Clock Domains 7–63
Unsupported Instance Assignments 7–64
Reviewing Conversion Results 7–64
Warning Messages 7–64
Ignored QSF Variable <assignment> 7–65
Global <name> = <value> 7–65
QSF: Expected <name> to be set to <expected value> but it is set to <actual value> 7–65
QSF: Found Global Fmax Requirement. Translation will be done using derive_clocks 7–65
TAN Report Database not found. HDL based assignments will not be migrated 7–65

Ignored Entity Assignment (Entity <entity>): <variable> = <value> -from <from> -to <to> 7–65
Ignoring OFFSET_FROM_BASE_CLOCK assignment for clock <clock> 7–66
Clock <clock> has no FMAX_REQUIREMENT - No clock was generated 7–66
No Clock Settings defined in QSF file 7–66
Clocks 7–66
Clock Transfers 7–66
Path Details 7–67
Unconstrained Paths 7–67
Bus Names 7–68
Other 7–68
Re-Running the Conversion Utility 7–68
Notes 7–68
Output Pin Load Assignments 7–68
Constraint Target Types 7–69
DDR Constraints with the DDR Timing Wizard 7–69
HardCopy Stratix Device Handoff 7–69
Unsupported SDC Features 7–69
Constraint Passing 7–70
Optimization 7–70
Clock Network Delay Reporting 7–70
PowerPlay Power Analysis 7–70
Project Management 7–71
Conversion Utility 7–71
t_{PD} and Minimum t_{PD} Requirement Conversion 7–71
Referenced Documents 7–72
Document Revision History 7–72

Chapter 8. Quartus II Classic Timing Analyzer

Introduction 8–1
Timing Analysis Tool Setup 8–2
Static Timing Analysis Overview 8–2
Clock Analysis 8–4
Clock Setup Check 8–4
Clock Hold Check 8–6
Multicycle Paths 8–7
Clock Settings 8–8
Individual Clock Settings 8–8
Default Clock Settings 8–8
Clock Types 8–9
Base Clocks 8–9
Derived Clocks 8–9
Undefined Clocks 8–9
PLL Clocks 8–10
Clock Uncertainty 8–11
Clock Latency 8–12
Timing Exceptions 8–15

Multicycle 8–15
Destination Multicycle Setup Exception 8–15
Destination Multicycle Hold Exception 8–16
Source Multicycle Setup Exception 8–17
Source Multicycle Hold Exception 8–18
Default Hold Multicycle 8–19
Clock Enable Multicycle 8–19
Setup Relationship and Hold Relationship 8–22
Maximum Delay and Minimum Delay 8–24
False Paths 8–24
I/O Analysis 8–26
External Input Delay and Output Delay Assignments 8–26
Input Delay Assignment 8–26
Output Delay Assignment 8–28
Virtual Clocks 8–29
Asynchronous Paths 8–30
Recovery and Removal 8–30
Recovery Report 8–31
Removal Report 8–32
Skew Management 8–34
Maximum Clock Arrival Skew 8–34
Maximum Data Arrival Skew 8–35
Generating Timing Analysis Reports with report_timing 8–36
Other Timing Analyzer Features 8–38
Wildcard Assignments 8–38
Assignment Groups 8–38
Fast Corner Analysis 8–40
Early Timing Estimation 8–40
Timing Constraint Checker 8–41
Latch Analysis 8–42
Timing Analysis Using the Quartus II GUI 8–43
Assignment Editor 8–43
Timing Settings 8–44
Clock Settings Dialog Box 8–45
More Timing Settings Dialog Box 8–46
Timing Reports 8–47
Advanced List Path 8–49
Early Timing Estimate 8–51
Assignment Groups 8–51
Scripting Support 8–52
Creating Clocks 8–53
Base Clocks 8–53
Derived Clocks 8–53
Clock Latency 8–53
Clock Uncertainty 8–54
Cut Timing Paths 8–54
Input Delay Assignment 8–54

Maximum and Minimum Delay 8–55
Maximum Clock Arrival Skew 8–55
Maximum Data Arrival Skew 8–55
Multicycle 8–56
Output Delay Assignment 8–56
Report Timing 8–57
Setup and Hold Relationships 8–57
Assignment Group 8–57
Virtual Clock 8–58
MAX+PLUS II Timing Analysis Methodology 8–58
f_{MAX} Relationships 8–58
Slack 8–58
I/O Timing 8–60
t_{SU} Timing 8–60
t_H Timing 8–60
t_{CO} Timing 8–61
Minimum t_{CO} (min t_{CO}) 8–62
t_{PD} Timing 8–62
Minimum t_{PD} (min t_{PD}) 8–62
The Timing Analyzer Tool 8–62
Conclusion 8–63
Referenced Documents 8–63
Document Revision History 8–64

Chapter 9. Synopsys PrimeTime Support

Introduction 9–1
Quartus II Settings for Generating the PrimeTime Software Files 9–2
Files Generated for the PrimeTime Software Environment 9–3
The Netlist 9–3
The SDO File 9–4
Generating Multiple Operating Conditions with TimeQuest 9–4
The Tcl Script 9–7
Generated File Summary 9–9
Running the PrimeTime Software 9–10
Analyzing Quartus II Projects 9–10
Other pt_shell Commands 9–11
PrimeTime Timing Reports 9–12
Sample of the PrimeTime Software Timing Report 9–12
Comparing Timing Reports from the Quartus II Classic Timing Analyzer and the PrimeTime Software 9–13
Clock Setup Relationship and Slack 9–13
Clock Hold Relationship and Slack 9–17
Input Delay and Output Delay Relationships and Slack 9–21
Static Timing Analyzer Differences 9–23
The Quartus II Classic Timing Analyzer and the PrimeTime Software 9–23
Rise/Fall Support 9–23
Minimum and Maximum Delays 9–23

Recovery/Removal Analysis 9–23
Encrypted Intellectual Property Blocks 9–24
Registered Clock Signals 9–24
Multiple Source and Destination Register Pairs 9–25
Latches 9–25
LVDS I/O 9–25
Clock Latency 9–26
Input and Output Delay Assignments 9–26
Generated Clocks Derived from Generated Clocks 9–26
The Quartus II TimeQuest Timing Analyzer and the PrimeTime Software 9–26
Encrypted Intellectual Property Blocks 9–26
Latches 9–27
LVDS I/O 9–27
The Quartus II TimeQuest Timing Analyzer SDC File and PrimeTime Compatibility . 9–27
Clock and Data Paths 9–27
Inverting and Non-Inverting Propagation 9–28
Multiple Rise/Fall Numbers For a Timing Arc 9–28
Virtual Generated Clocks 9–28
Generated Clocks Derived from Generated Clocks 9–28
Conclusion 9–28
Referenced Documents 9–29
Document Revision History 9–30

Section III. Power Estimation and Analysis

Chapter 10. PowerPlay Power Analysis

Introduction 10–1
Quartus II Early Power Estimator File 10–2
PowerPlay Early Power Estimator File Generator Compilation Report 10–5
Types of Power Analyses 10–6
Factors Affecting Power Consumption 10–6
Device Selection 10–6
Environmental Conditions 10–7
Air Flow 10–7
Heat Sink and Thermal Compound 10–7
Ambient Temperature 10–8
Board Thermal Model 10–8
Design Resources 10–8
Number, Type, and Loading of I/O Pins 10–8
Number and Type of Logic Elements, Multiplier Elements, and RAM Blocks 10–8
Number and Type of Global Signals 10–9
Signal Activities 10–9
PowerPlay Power Analyzer Flow 10–10
Operating Conditions 10–11
Signal Activities Data Sources 10–12

Simulation Results 10–13
Using Simulation Files in Modular Design Flows 10–15
Complete Design Simulation 10–16
Modular Design Simulation 10–16
Multiple Simulations on the Same Entity 10–17
Overlapping Simulations 10–18
Partial Simulations 10–18
Node Name Matching Considerations 10–18
Glitch Filtering 10–19
Node and Entity Assignments 10–21
Timing Assignments to Clock Nodes 10–22
Default Toggle Rate Assignment 10–22
Vectorless Estimation 10–23
Using the PowerPlay Power Analyzer 10–23
Common Analysis Flows 10–23
Signal Activities from Full Post-Fit Netlist (Timing) Simulation 10–23
Signal Activities from RTL (Functional) Simulation, Supplemented by Vectorless Estimation 10–24
Signal Activities from Vectorless Estimation, User-Supplied Input Pin Activities 10–24
Signal Activities from User Defaults Only 10–24
Generating a SAF or VCD File Using the Quartus II Simulator 10–24
Generating a VCD File Using a Third-Party Simulator 10–28
Running the PowerPlay Power Analyzer Using the Quartus II GUI 10–31
PowerPlay Power Analyzer Compilation Report 10–39
Summary 10–39
Settings 10–39
Simulation Files Read 10–39
Operating Conditions Used 10–39
Thermal Power Dissipated by Block 10–39
Thermal Power Dissipation by Block Type (Device Resource Type) 10–39
Thermal Power Dissipation by Hierarchy 10–40
Core Dynamic Thermal Power Dissipation by Clock Domain 10–40
Current Drawn from Voltage Supplies 10–40
Confidence Metric Details 10–40
Signal Activities 10–41
Messages 10–41
Specific Rules for Reporting 10–41
Scripting Support 10–41
Running the PowerPlay Power Analyzer from the Command Line 10–42
Conclusion 10–43
Referenced Documents 10–43
Document Revision History 10–44

Section IV. Signal Integrity

Chapter 11. Signal Integrity Analysis with Third-Party Tools

Introduction 11–1
The Need for FPGA to Board Signal Integrity Analysis 11–3
The Double Counting Problem for FPGA Output Timing 11–4
Defining the Double Counting Problem 11–4
The Solution to Double Counting 11–5
I/O Model Selection: IBIS or HSPICE 11–7
FPGA to Board Signal Integrity Analysis Flow 11–8
Create I/O and Board Trace Model Assignments 11–10
Enable Output File Generation 11–10
Generate the Output Files 11–10
Customize the Output Files 11–11
Set Up and Run Simulations in Third-Party Tools 11–11
Interpret Simulation Results 11–12
Simulation with IBIS Models 11–12
Elements of an IBIS Model 11–12
Creating Accurate IBIS Models 11–13
Download IBIS Models 11–13
Generate Custom IBIS Models with the IBIS Writer 11–14
Design Simulation Using the Mentor Graphics HyperLynx Software 11–17
Configuring LineSim to Use Altera IBIS Models 11–20
Integrating Altera IBIS Models into LineSim Simulations 11–21
Running and Interpreting LineSim Simulations 11–24
Simulation with HSPICE Models 11–25
Supported Devices and Signaling 11–26
Creating Accurate HSPICE Models 11–26
Creating HSPICE Model Files Using the Quartus II GUI 11–27
Creating HSPICE Model Files Using Tcl Scripting and the Command Line 11–28
Customizing HSPICE Model Files 11–29
Design Simulation Using Synopsys HSPICE 11–30
Running HSPICE Simulations 11–30
Viewing and Interpreting Tabular Simulation Results 11–31
Viewing Graphical Simulation Results 11–31
Making Design Adjustments Based on HSPICE Simulations 11–33
Conclusion 11–35
Referenced Documents 11–36
Document Revision History 11–36

Section V. In-System Design Debugging

Chapter 12. Quick Design Debugging Using SignalProbe

Introduction 12–1

On-Chip Debugging Tool Comparison 12–2
Debugging Using the SignalProbe Feature 12–4
Reserve the SignalProbe Pins 12–4
Perform a Full Compilation 12–6
Assign a SignalProbe Source 12–6
Add Registers to the Pipeline Path to SignalProbe Pin 12–7
Perform a SignalProbe Compilation 12–9
Analyze the Results of the SignalProbe Compilation 12–10
Generate the Programming File 12–11
SignalProbe ECO flows 12–11
SignalProbe ECO Flow with Quartus II Incremental Compilation 12–11
SignalProbe ECO Flow without Quartus Incremental Compilation 12–12
Common Questions About the SignalProbe Feature 12–14
Why Did I Get the Following Error Message, "Error: There are No Enabled SignalProbes to Process"? 12–14
How Can I Retain My SignalProbe ECOs during Re-compilation of My Design? 12–14
Why Did My SignalProbe Source Disappear in the Change Manager? 12–14
What is an ECO and Where Can I Find More Information on ECO? 12–15
How Do I Migrate My Previous SignalProbe Assignments in the Quartus II Software Versions 5.1 and below to Versions 6.0 and Higher? 12–15
What are all the Changes for the SignalProbe Feature between the Quartus II Software Version 5.1 and Earlier, and Version 6.0 and Later? 12–16
Scripting Support 12–17
Make a SignalProbe Pin 12–17
Delete a SignalProbe Pin 12–17
Enable a SignalProbe Pin 12–18
Disable a SignalProbe Pin 12–18
Perform a SignalProbe Compilation 12–18
Migrating Previous SignalProbe Pins to the Quartus II Software Versions 6.0 and Later 12–18
Script Example 12–18
Using SignalProbe with the APEX Device Family 12–19
Adding SignalProbe Sources 12–19
Performing a SignalProbe Compilation 12–20
Running SignalProbe with Smart Compilation 12–21
Understanding the Results of a SignalProbe Compilation 12–21
Analyzing SignalProbe Routing Failures 12–23
SignalProbe Scripting Support for APEX Devices 12–23
Reserving SignalProbe Pins 12–24
Adding SignalProbe Sources 12–24
Assigning I/O Standards 12–24
Adding Registers for Pipelining 12–24
Run SignalProbe Automatically 12–25
Run SignalProbe Manually 12–25
Enable or Disable All SignalProbe Routing 12–25
Running SignalProbe with Smart Compilation 12–26
Allow SignalProbe to Modify Fitting Results 12–26

Conclusion 12–26
Referenced Documents 12–26
Document Revision History 12–27

Chapter 13. Design Debugging Using the SignalTap II Embedded Logic Analyzer

Introduction 13–1
Hardware and Software Requirements 13–3
On-Chip Debugging Tool Comparison 13–5
Design Flow Using the SignalTap II Logic Analyzer 13–7
SignalTap II Logic Analyzer Task Flow 13–8
Add the SignalTap II Logic Analyzer to Your Design 13–9
Configure the SignalTap II Logic Analyzer 13–9
Define Triggers 13–9
Compile the Design 13–9
Program the Target Device or Devices 13–9
Run the SignalTap II Logic Analyzer 13–10
View, Analyze, and Use Captured Data 13–10
Add the SignalTap II Logic Analyzer to Your Design 13–10
Creating and Enabling a SignalTap II File 13–10
Creating a SignalTap II File 13–10
Enabling and Disabling a SignalTap II File for the Current Project 13–11
Using the MegaWizard Plug-In Manager to Create Your Embedded Logic Analyzer 13–12
Creating an HDL Representation Using the MegaWizard Plug-In Manager 13–12
SignalTap II Megafunction Ports 13–16
Instantiating the SignalTap II Logic Analyzer in Your HDL 13–16
Embedding Multiple Analyzers in One FPGA 13–17
Monitoring FPGA Resources Used by the SignalTap II Logic Analyzer 13–17
Configure the SignalTap II Logic Analyzer 13–18
Assigning an Acquisition Clock 13–18
Adding Signals to the SignalTap II File 13–19
Signal Preservation 13–21
Assigning Data Signals 13–22
Node List Signal Use Options 13–23
Untappable Signals 13–23
Adding Signals with a Plug-In 13–23
Specifying the Sample Depth 13–25
Capturing Data to a Specific RAM Type 13–26
Choosing the Buffer Acquisition Mode 13–26
Circular Buffer 13–27
Segmented Buffer 13–27
Managing Multiple SignalTap II Files and Configurations 13–28
Define Triggers 13–30
Creating Basic Trigger Conditions 13–30
Creating Advanced Trigger Conditions 13–31
Examples of Advanced Triggering Expressions 13–32
Trigger Condition Flow Control 13–34
Sequential Triggering 13–34

Custom State-Based Triggering 13–36
State Diagram Pane 13–39
State Machine Pane 13–39
Resources Pane 13–39
SignalTap II Trigger Flow Description Language 13–40
State Labels 13–41
Boolean_expression 13–41
Action_list 13–42
Resource Manipulation Action 13–43
Buffer Control Action 13–43
State Transition Action 13–44
Specifying the Trigger Position 13–44
Creating a Power-Up Trigger 13–45
Enabling a Power-Up Trigger 13–45
Managing and Configuring Power-Up and Runtime Trigger Conditions 13–46
Using External Triggers 13–47
Trigger In 13–47
Trigger Out 13–48
Using the Trigger Out of One Analyzer as the Trigger In of Another Analyzer 13–48
Compile the Design 13–50
Faster Compilations with Quartus II Incremental Compilation 13–51
Enabling Incremental Compilation for your Design 13–51
Using Incremental Compilation with the SignalTap II Logic Analyzer 13–52
Preventing Changes Requiring Recompilation 13–54
Timing Preservation with the SignalTap II Logic Analyzer 13–54
Performance and Resource Considerations 13–55
Program the Target Device or Devices 13–57
Programming a Single Device 13–57
Programming Multiple Devices to Debug Multiple Designs 13–58
Run the SignalTap II Logic Analyzer 13–59
Running with a Power-Up Trigger 13–60
Running with Runtime Triggers 13–60
Performing a Force Trigger 13–61
SignalTap II Status Messages 13–62
View, Analyze, and Use Captured Data 13–63
Viewing Captured Data 13–63
Creating Mnemonics for Bit Patterns 13–64
Automatic Mnemonics with a Plug-In 13–64
Locating a Node in the Design 13–65
Saving Captured Data 13–66
Converting Captured Data to Other File Formats 13–66
Creating a SignalTap II List File 13–67
Other Features 13–67
Using the SignalTap II MATLAB MEX Function to Capture Data 13–67
Using SignalTap II in a Lab Environment 13–69
Remote Debugging Using the SignalTap II Logic Analyzer 13–69
Equipment Setup 13–70

Software Setup on the Remote PC 13–70
Software Setup on the Local PC 13–71
SignalTap II Setup on the Local PC 13–72
SignalTap II Scripting Support 13–72
SignalTap II Command Line Options 13–73
SignalTap II Tcl Commands 13–75
Design Example: Using SignalTap II Logic Analyzers in SOPC Builder Systems 13–77
Custom Triggering Flow Application Examples 13–77
Design Example 1: Specifying a Custom Trigger Position 13–77
Design Example 2: Trigger When triggercond1 Occurs Ten Times between triggercond2 and triggercond3 13–78
Conclusion 13–80
Referenced Documents 13–80
Document Revision History 13–81

Chapter 14. In-System Debugging Using External Logic Analyzers

Introduction 14–1
Choosing a Logic Analyzer 14–2
Required Components 14–3
FPGA Device Support 14–3
Debugging Your Design Using the Logic Analyzer Interface 14–4
Creating a Logic Analyzer Interface File 14–4
Creating a New Logic Analyzer Interface File 14–5
Opening an Existing External Analyzer Interface File 14–6
Saving the External Analyzer Interface File 14–7
Configuring the Logic Analyzer Interface File Core Parameters 14–7
Mapping the Logic Analyzer Interface File Pins to Available I/O Pins 14–9
Mapping Internal Signals to the Logic Analyzer Interface Banks 14–9
Using the Node Finder 14–10
Enabling the Logic Analyzer Interface Before Compiling Your Quartus II Project 14–11
Compiling Your Quartus II Project 14–12
Programming Your FPGA Using the Logic Analyzer Interface 14–13
Using the Logic Analyzer Interface with Multiple Devices 14–14
Configuring Banks in the Logic Analyzer Interface File 14–15
Acquiring Data on Your Logic Analyzer 14–15
Advanced Features 14–15
Using the Logic Analyzer Interface with Incremental Compilation 14–15
Creating Multiple Logic Analyzer Interface Instances in One FPGA 14–16
Conclusion 14–17
Document Revision History 14–18

Chapter 15. In-System Updating of Memory and Constants

Introduction 15–1
Overview 15–1
Device Megafunction Support 15–2
Using In-System Updating of Memory Constants with Your Design 15–3
Creating In-System Modifiable Memories Constants 15–3

Running the In-System Memory Content Editor 15–4
Instance Manager 15–5
Editing Data Displayed in the Hex Editor 15–7
Importing Exporting Memory Files 15–7
Viewing Memories Constants in the Hex Editor 15–7
Scripting Support 15–9
Programming the Device Using the In-System Memory Content Editor 15–10
Example: Using the In-System Memory Content Editor with the SignalTap II Embedded Logic Analyzer 15–10
Conclusion 15–11
Referenced Documents 15–11
Document Revision History 15–12

Chapter 16. Design Debugging Using In-System Sources and Probes

Introduction 16–1
Overview 16–1
Hardware and Software Requirements 16–3
Design Flow Using In-System Sources and Probes 16–4
Configuring the altsource_probes Megafunction 16–6
Instantiating the altsource_probe Megafunction 16–8
Compiling the Design 16–8
Running the
In-System Sources and Probes Editor 16–9
Programming Your Device Using the JTAG Chain Configuration Window 16–11
Instance Manager 16–12
Sources and Probes Editor Window 16–13
Reading Probe Data 16–13
Writing Data 16–13
Data Organization 16–14
TCL Support 16–14
Design Example: Dynamic PLL Reconfiguration 16–18
Conclusion 16–21
Referenced Documents 16–21
Document Revision History 16–21

Section VI. Formal Verification

Chapter 17. Cadence Encounter Conformal Support

Introduction 17–1
Formal Verification Versus Simulation 17–2
Formal Verification: What You Need to Know 17–2
Formal Verification Design Flow 17–2
Quartus II Integrated Synthesis 17–3
EDA Tool Support for Quartus II Integrated Synthesis 17–3
Synplify Pro 17–3

EDA Tool Support for Synplify Pro 17–4
RTL Coding Guidelines for Quartus II Integrated Synthesis 17–5
Synthesis Directives and Attributes 17–5
Stuck-at Registers 17–7
ROM, LPM_DIVIDE, and Shift Register Inference 17–8
RAM Inference 17–8
Latch Inference 17–9
Combinational Loops 17–9
Finite State Machine Coding Styles 17–10
Generating the Post-Fit Netlist Output File and the Encounter Conformal Setup Files 17–10
Tcl Command 17–15
GUI 17–15
The Quartus II Software Generated Files, Formal Verification Scripts, and Directories ... 17–16
Understanding the Formal Verification Scripts for Encounter Conformal 17–18
The Encounter Conformal Commands within the Quartus II Software-Generated Scripts .....
.....17–18
Comparing Designs Using Encounter Conformal 17–21
Black Boxes in the Encounter Conformal Flow 17–21
Running the Encounter Conformal Software 17–22
Running the Encounter Conformal Software from the GUI 17–22
Running the Encounter Conformal Software From a System Command Prompt 17–24
Known Issues and Limitations 17–24
Conclusion 17–27
Black Box Models 17–28
Conformal Dofile/Script Example 17–30
Referenced Documents 17–32
Document Revision History 17–33

Chapter 18. Synopsys Formality Support

Introduction 18–1
Formal Verification 18–1
Equivalence Checking 18–1
Formal Verification Support 18–2
EDA Tools and Device Support 18–2
Formal Verification Between RTL and Post-Synthesis Netlist 18–2
Generating Post-Synthesis Netlist for Formal Verification 18–3
DC FPGA Software Settings 18–3
Generating the VO File and Formality Script 18–4
Handling Black Boxes 18–9
Tcl Command 18–9
GUI 18–10
Quartus II Scripts for Formality 18–11
Comparing Designs Using the Formality Software 18–11
Known Issues and Limitations 18–12
Conclusion 18–12
Related Links 18–12
Tcl Sample Script 18–13

DC FPGA Synthesis Script 18–13
Quartus II Software-Generated Formal Verification Script 18–14
Referenced Documents 18–15
Document Revision History 18–15

Section VII. Device Programming

Chapter 19. Quartus II Programmer

Introduction 19–1
Programming Flow 19–1
Programming and Configuration Modes 19–4
JTAG Mode 19–4
Passive Serial Mode 19–4
Active Serial Mode 19–5
In-Socket Programming Mode 19–5
Programmer Overview 19–6
Tools Menu 19–11
Hardware Setup 19–12
Hardware Settings 19–12
JTAG Settings 19–13
Device Programming and Configuration 19–14
Single Device Programming and Configuration 19–14
Multi-Device Programming and Configuration 19–14
Bypassing an Altera Device 19–15
Bypassing a Non-Altera Device 19–15
Chain Description File 19–17
Design Security Key Programming 19–17
Optional Programming Files 19–18
Types of Programming and Configuration Files 19–18
Generating Optional Programming Files 19–20
Create Programming Files 19–20
Convert Programming Files 19–20
Generating Optional Programming or Configuration Files During Compilation 19–21
Flash Loaders 19–21
Parallel Flash Loader 19–21
Serial Flash Loader 19–21
Other Programming Tools 19–22
Quartus II Stand-Alone Programmer 19–22
jtagconfig Debugging Tool 19–22
Scripting Support 19–22
Conclusion 19–23
Referenced Documents 19–24
Document Revision History 19–24

Chapter Revision Dates

The chapters in this book, the Quartus II Handbook, Volume 3, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1. Quartus II Simulator
Revised: *October 2007*
Part number: *QII53017-7.2.0*

Chapter 2. Mentor Graphics ModelSim Support
Revised: *October 2007*
Part number: *QII53001-7.2.0*

Chapter 3. Synopsys VCS Support
Revised: *October 2007*
Part number: *QII53002-7.2.0*

Chapter 4. Cadence NC-Sim Support
Revised: *October 2007*
Part number: *QII53003-7.2.0*

Chapter 5. Simulating Altera IP in Third-Party Simulation Tools
Revised: *October 2007*
Part number: *QII53014-7.2.0*

Chapter 6. The Quartus II TimeQuest Timing Analyzer
Revised: *October 2007*
Part number: *QII53018-7.2.0*

Chapter 7. Switching to the Quartus II TimeQuest Timing Analyzer
Revised: *October 2007*
Part number: *QII53019-7.2.0*

Chapter 8. Quartus II Classic Timing Analyzer
Revised: *October 2007*
Part number: *QII53004-7.2.0*

Chapter 9. Synopsys PrimeTime Support
Revised: *October 2007*
Part number: *QII53005-7.2.0*

Chapter 10. PowerPlay Power Analysis
Revised: *October 2007*
Part number: *QII53013-7.2.0*

Chapter 11. Signal Integrity Analysis with Third-Party Tools
Revised: *October 2007*
Part number: *QII53020-7.2.0*

Chapter 12. Quick Design Debugging Using SignalProbe
Revised: *October 2007*
Part number: *QII53008-7.2.0*

Chapter 13. Design Debugging Using the SignalTap II Embedded Logic Analyzer
Revised: *October 2007*
Part number: *QII53009-7.2.0*

Chapter 14. In-System Debugging Using External Logic Analyzers
Revised: *October 2007*
Part number: *QII53016-7.2.0*

Chapter 15. In-System Updating of Memory and Constants
Revised: *October 2007*
Part number: *QII53012-7.2.0*

Chapter 16. Design Debugging Using In-System Sources and Probes
Revised: *October 2007*
Part number: *QII53021-7.2.0*

Chapter 17. Cadence Encounter Conformal Support
Revised: *October 2007*
Part number: *QII53011-7.2.0*

Chapter 18. Synopsys Formality Support
Revised: *October 2007*
Part number: *QII53015-7.2.0*

Chapter 19. Quartus II Programmer
Revised: *October 2007*
Part number: *QII53022-7.2.0*

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 7.2.

How to Contact Altera

For the most up-to-date information about Altera products, refer to the following table.

Information Type	Contact *(1)*
Technical support	www.altera.com/mysupport/
Technical training	www.altera.com/training/ custrain@altera.com
Product literature	www.altera.com/literature/
Altera literature services	literature@altera.com *(1)*
FTP site	ftp.altera.com

Note to table:
(1) You can also contact your local Altera sales office or sales representative.

Third-Party Software Product Information

Third-party software products described in this handbook are not Altera products, are licensed by Altera from third parties, and are subject to change without notice. Updates to these third-party software products may not be concurrent with Quartus II software releases. Altera has assumed responsibility for the selection of such third-party software products and its use in the Quartus II 7.2 software release. To the extent that the software products described in this handbook are derived from third-party software, no third party warrants the software, assumes any liability regarding use of the software, or undertakes to furnish you any support or information relating to the software. EXCEPT AS EXPRESSLY SET FORTH IN THE APPLICABLE ALTERA PROGRAM LICENSE SUBSCRIPTION AGREEMENT UNDER WHICH THIS SOFTWARE WAS PROVDED TO YOU, ALTERA AND THIRD-PARTY LICENSORS DISCLAIM ALL WARRANTIES WITH RESPECT TO THE USE OF SUCH THIRD-PARTY SOFTWARE CODE OR DOCUMENTATION IN THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT. For more information, including the latest available version of specific third-party software products, refer to the documentation for the software in question.

Typographic Conventions

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
✓, —, N/A	Used in table cells to indicate the following: ✓ indicates a "Yes" or "Applicable" statement; — indicates a "No" or "Not Supported" statement; N/A indicates that the table cell entry is not applicable to the item of interest.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.

Section I. Simulation

As the design complexity of FPGAs continues to rise, verification engineers are finding it increasingly difficult to simulate their system-on-a-programmable-chip (SOPC) designs in a timely manner. The verification process is now the bottleneck in the FPGA design flow. You can perform functional and timing simulation of your design by using the Quartus® II Simulator. The Quartus II software also provides a wide range of features for performing simulation of designs in EDA simulation tools.

This section includes the following chapters:

- Chapter 1, Quartus II Simulator
- Chapter 2, Mentor Graphics ModelSim Support
- Chapter 3, Synopsys VCS Support
- Chapter 4, Cadence NC-Sim Support
- Chapter 5, Simulating Altera IP in Third-Party Simulation Tools



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

1. Quartus II Simulator

QII53017-7.2.0

Introduction

With today's FPGAs becoming faster and more complex, designers face challenges in validating their designs. Simulation verifies the correctness of the design, reducing board testing and debugging time.

Altera® offers the Simulator as part of the Quartus® II software to assist designers with design verification. The Quartus II Simulator has a comprehensive set of features that are covered in the following sections:

- "Simulation Flow"
- "Waveform Editor" on page 1–5
- "Simulator Settings" on page 1–17
- "Simulation Report" on page 1–25
- "Debugging with the Quartus II Simulator" on page 1–29
- "Scripting Support" on page 1–32

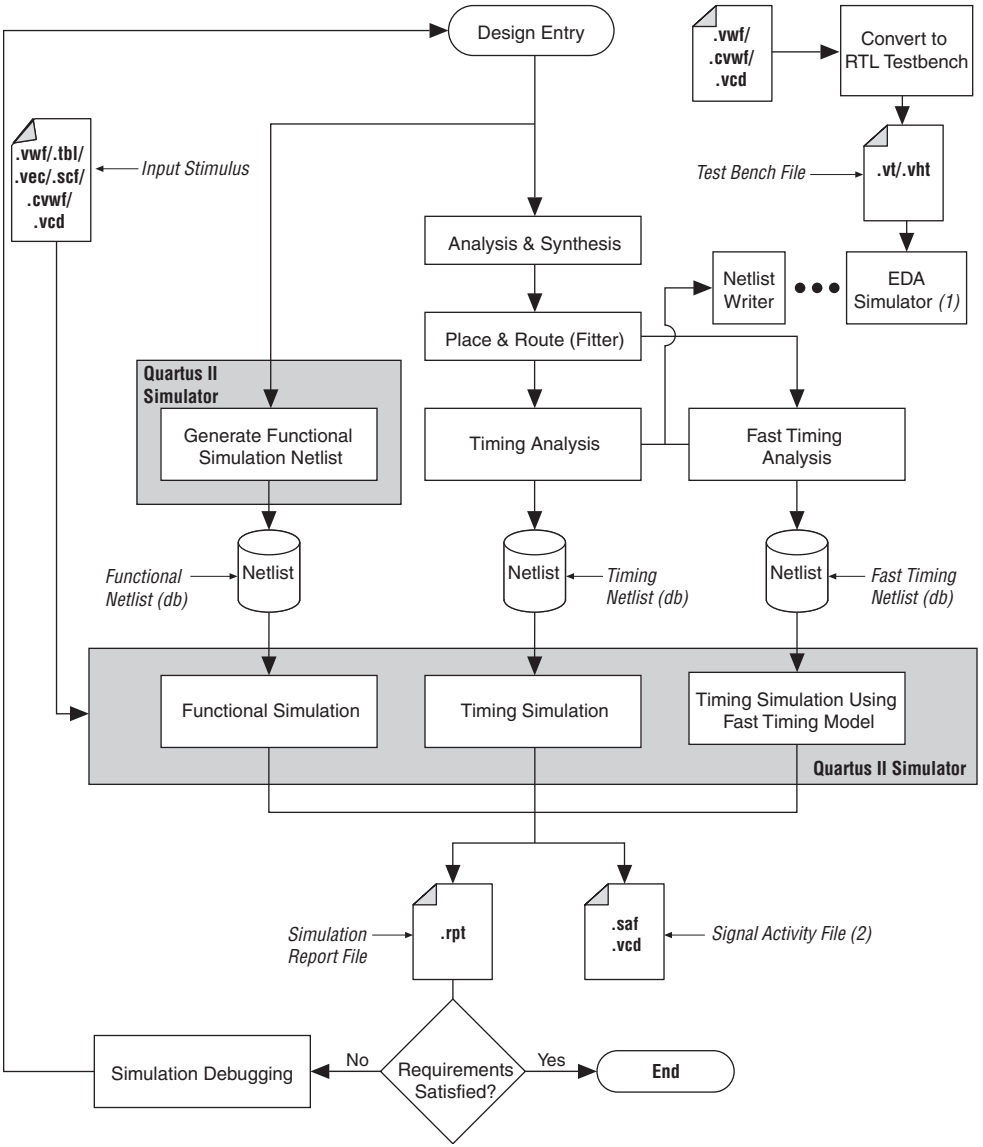
This chapter describes how to perform different types of simulations with the Quartus II Simulator.

Simulation Flow

You can perform both functional and timing simulations with the Quartus II Simulator. Both types of simulation verify the correctness and behavior of your design. Functional simulations are run at the beginning of the Quartus II design flow and timing simulations are run at the end.

Figure 1–1 shows the Quartus II Simulator flow.

Figure 1–1. Simulation Flow



Notes to Figure 1–1:

(1) For more information on EDA Simulators, refer to the *Simulation* section in volume 3 of the *Quartus II Handbook*.

(2) You can use Signal Activity Files (**.saf**) or Value Change Dump Files (**.vcd**) in the PowerPlay Power Analyzer to check power resources.

As shown in Figure 1–1, your design simulation can happen at the functional level, where your design's logical behavior is verified and no timing information is used in simulation. Timing simulation can happen after your design has been compiled (synthesized and placed and routed) and after you use the timing data of your design's resources. In Timing simulation, your design's logical behavior is verified with the device's worst-case timing models. Timing simulation using the Fast Timing Model is also a type of Timing simulation where best-case timing data is used.

To perform functional simulations with the Quartus II Simulator, you must first generate a functional simulation netlist. A functional netlist file is a flattened netlist extracted from the design files that does not contain timing information.

For timing simulations, you must first perform place-and-route and static timing analysis to generate a timing simulation netlist. A timing simulation netlist includes timing delays of each device atom block and the routing delays.

If you want to use third-party EDA simulation tools, you can generate a netlist using EDA Netlist Writer. You can use this netlist with your testbench files in third-party simulation tools.



For more information about third-party simulators, refer to the respective EDA Simulation chapter in the *Simulation* section in volume 3 of the *Quartus II Handbook*.

The Quartus II Simulator supports Functional, Timing, and Timing using Fast Timing Model simulations. The following sections describe how to perform these simulations.

Functional Simulation

To run a functional simulation, perform the following steps:

1. On the Processing menu, click **Generate Functional Simulation Netlist**. This flattens the functional simulation netlist extracted from the design files. The netlist does not contain timing information.
2. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
3. In the **Category** list, select **Simulator Settings**. The **Simulator Settings** page appears.
4. In the **Simulation mode** list, select **Functional**.

5. In the **Simulation input** box, specify the vector source. You must specify the vector file to run the simulation.

6. Click **OK**.

7. On the Processing menu, click **Start Simulation**.

Timing Simulation

To run a timing simulation, perform the following steps:

1. On the Processing menu, click **Start Compilation** or click the **Compilation** button on the toolbar. This flattens the design and generates an internal netlist with timing delay information annotated.

2. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.

3. In the **Category** list, select **Simulator Settings**. The **Simulator Settings** page appears.

4. In the **Simulation Mode** list, select **Timing**.

5. In the **Simulation input** list, specify the vector source. You need to specify the vector file to run the simulation.

6. Click **OK**.

7. On the Processing menu, click **Start Simulation**.

Timing Simulation Using Fast Timing Model Simulation

To run a timing simulation using a fast timing model, perform the following steps:

1. On the Processing menu, point to Start and click **Start Analysis and Synthesis**.

2. On the Processing menu, point to Start and click **Start Fitter**.

   You must perform fast timing analysis before you can perform a timing simulation using the fast timing models.

3. On the Processing menu, point to Start and click **Start Classic Timing Analyzer (Fast Timing Model)**.

4. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
5. In the **Category** list, select **Simulator Settings**. The **Simulator Settings** page appears.
6. In the **Simulation mode** list, select **Timing using Fast Timing Model**.
7. In the **Simulation input** box, specify the vector source. You need to specify the vector file to run the simulation.
8. Click **OK**.
9. On the Processing menu, click **Start Simulation**.

Waveform Editor

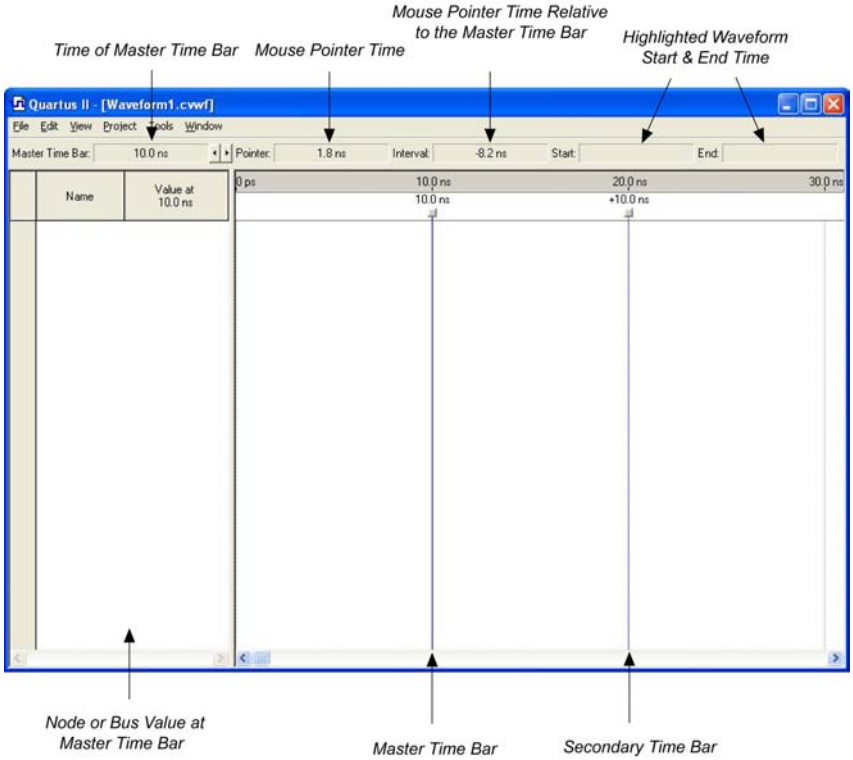
The most common input stimulus for the Quartus II Simulator are VWFs. You can use the Quartus II Waveform Editor to generate a VWF.

Creating VWFs

To create a VWF, perform the following steps:

1. On the File menu, click **New**. The **New** dialog box appears.
2. Click the **Other Files** tab, and select **Vector Waveform File**.
3. Click **OK**. A blank Waveform Editor window appears (Figure 1–2).

Figure 1–2. Waveform Editor Window

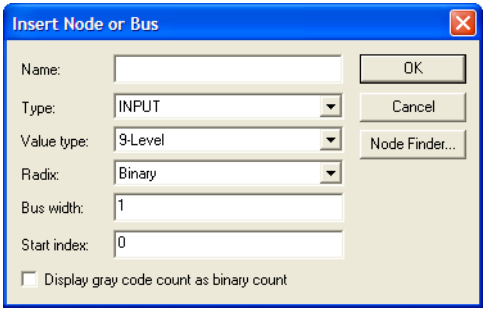


4. Add nodes and buses. To add a node or bus, on the Edit menu, click **Insert** and click **Insert Node or Bus**. The **Insert Node or Bus** dialog box appears (Figure 1–3). All nodes and buses, as well as the internal signals, are listed under **Name** in the Waveform Editor window.



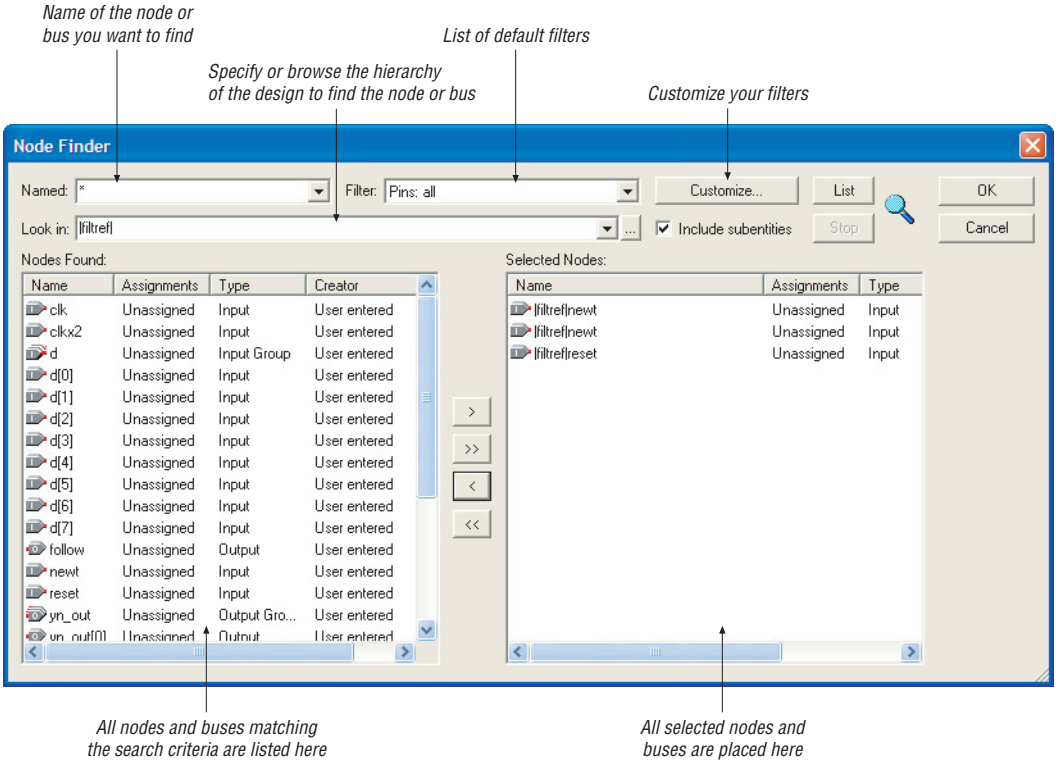
You can also open the **Insert Node or Bus** dialog box by double-clicking under **Name** in the Waveform Editor.

Figure 1–3. Insert Node or Bus Dialog Box



5. You can customize the type of node or bus you want to add. If you have a large design with many nodes or buses, you may want to use the Node Finder for node or bus selection. To use the Node Finder, click **Node Finder**. The **Node Finder** dialog box appears (Figure 1–4).

Figure 1–4. Node Finder Dialog Box



You can use the Node Finder to find your nodes for simulation among all the nodes and buses in your design. Use the Node Finder to filter and add nodes to your waveform. The Node Finder is equipped with multiple default filter options. By using the correct filter in the Node Finder, you can find the internal node's name and add it to your Vector Waveform File for simulation.

Your node might not appear in the simulation waveform and might be ignored during simulation. This happens because the node has been renamed or synthesized away by the Quartus II software. To prevent this from happening, Altera recommends using the register and pin nodes to simulate your design.

Table 1–1 describes twelve of the Node Finder default filters.

Table 1–1. Filter Options

Filter	Description
Pins: input	Finds all input pin names in your design file(s).
Pins: output	Finds all output pin names in your design file(s).
Pins: bidirectional	Finds all bidirectional pin names in your design file(s).
Pins: virtual	Finds all virtual pin names.
Pins: all	Finds all pin names in your design file(s).
Registers: pre-synthesis	Finds all user-entered register names contained in the design after design elaboration, but before physical synthesis does any synthesis optimizations.
Registers: post-fitting	Finds all user-entered register names in your design file(s) that survived physical synthesis and fitting.
Design Entry (all names)	Finds all user-entered names in your design file(s).
Post-Compilation	Finds all user-entered and compiler-generated names that do not have location assignments and survived fitting.
SignalTap II: pre-synthesis	Finds all internal device nodes in the pre-synthesis netlist that can be analyzed by the SignalTap® II Logic Analyzer.
SignalTap II: post-fitting	Finds all internal device nodes in the post-fitting netlist that can be analyzed by the SignalTap II Logic Analyzer.
SignalProbe	Finds all SignalProbe™ device nodes in the post-fitting netlist.

To customize your own filters in the Node Finder, perform the following steps:

a. Click **Customize**. The **Customize Filter** dialog box appears.

b. To configure settings, click **New**. The **New Custom Filter** dialog box appears.

c. In the **Filter name** box, type the name of the custom filter.

d. In the **Copy settings from filter** list, select the filter setting.

e. Click **OK**.

f. You can now customize your filters in the **Customize Filter** dialog box.

6. In the **Look in** box, you can view and edit the current search hierarchy path. You can type the search hierarchy path or you can browse for the hierarchy path by clicking the browse button.

You can move up the search hierarchy by selecting hierarchical names in the **Select Hierarchy Level** dialog box. This ensures that in a large design with many signals, you can specify which hierarchy you would like to get the node from to reduce the amount of signals displayed.

7. After you have configured the filter and specified the correct hierarchy in the **Node Finder** dialog box, click **List** to display all relevant nodes or buses.

   Select any node(s) or bus(es) from the **Nodes Found** list and click **>** to include it in the waveform, or you can click **>>** to include all nodes and buses displayed in the **Nodes Found** list.

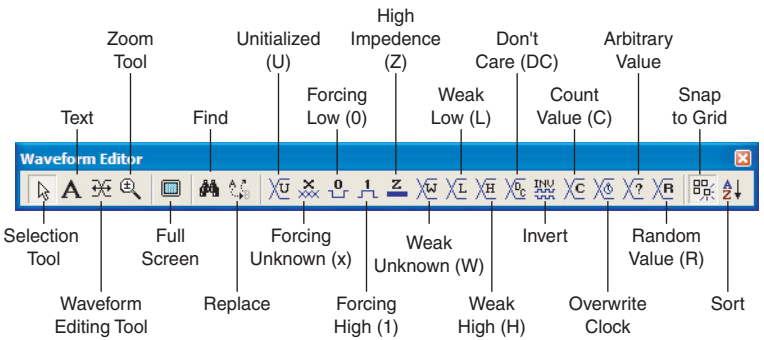
8. Click **OK**.



You can also add nodes to the Waveform Editor by dragging nodes from the Project Navigator, Netlist Viewers, or Block Diagram, and dropping them into the Waveform Editor.

9. Create a waveform for a signal. The Quartus II Waveform Editor toolbar includes some of the most common waveform settings, making waveform vector drawings easier and user friendly. Figure 1–5 shows the options available on the Waveform Editor toolbar.

Figure 1–5. Waveform Editor Toolbar



10. After you edit your waveform, save the waveform. On the File menu, click **Save As**. The **Save As** dialog box appears. Type your file name, specify the file type, and click **Save**.

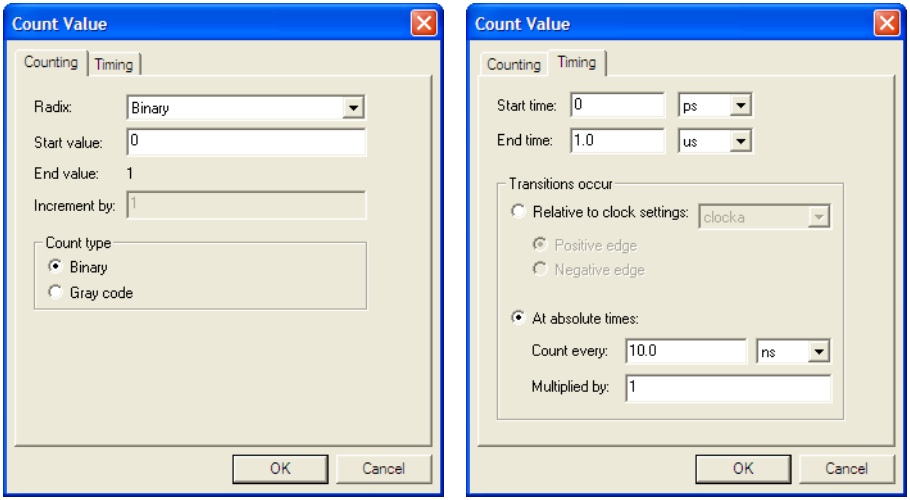


Instead of using the Node Finder to insert your nodes for your VWF, you can also drag-and-drop any nodes from the Netlist Viewer to your Simulation Vector Waveform File. For more information on Netlist Viewers, refer to *Analyzing Designs with the Quartus II Netlist Viewers* in volume 1 of the *Quartus II Handbook*.

Count Value

Count Value applies a count value to a bus to increment the value of the bus by a specified time interval. Instead of manually editing the values for each node, the Count Value feature on the Waveform Editor toolbar automatically creates the counting values for buses. This feature enables you to specify a starting value for a bus, what time interval to increment, and when to stop counting. You can also configure transition occurrences, while setting the count type and increment number. When you click on the **Count Value** button in the Waveform Editor toolbar, the **Count Value** dialog box appears (Figure 1–6). You can also open the **Count Value** dialog box by right-clicking the selected node, pointing to Value, and clicking **Count Value**.

Figure 1–6. Count Value Dialog Box

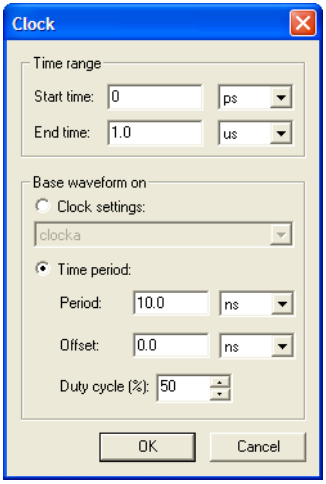


Clock

You can use the Clock feature in the Waveform Editor toolbar to automatically generate the clock wave, rather than drawing each clock triggering pulse. To generate a clock signal with the **Clock** dialog box,

click the **Overwrite Clock** button on the Waveform Editor toolbar. Furthermore, you can determine the start and end time of a clock signal, whether to manually configure the period (the offset and the duty cycle), or whether to generate the clock based on a specified clock. Figure 1–7 shows the **Clock** dialog box.

Figure 1–7. Clock Dialog Box

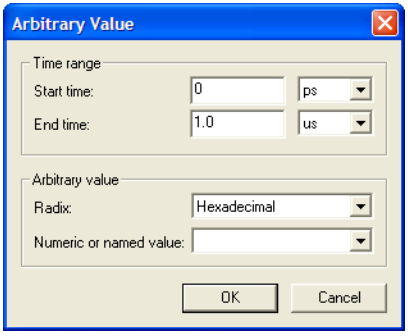


Arbitrary Value

Arbitrary Value allows you to overwrite a node value over the selected waveform, waveform interval, or across one or more nodes or groups. To overwrite a node value, perform the following steps:

1. Select a node or a bus and click the **Arbitrary Value** button on the Waveform Editor toolbar (Figure 1–5). The **Arbitrary Value** dialog box appears (Figure 1–8).
2. Under **Time range**, specify the start and end time you want to overwrite for the node value.
3. In the **Radix** list, select the radix type.
4. Specify the new value you want overwritten in the **Numeric or named value** box.
5. Click **OK**.

Figure 1–8. Arbitrary Value Dialog Box

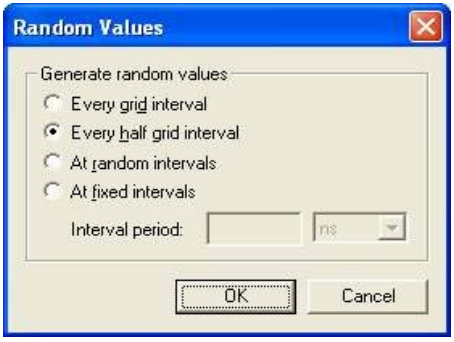


Random Value

Random Value allows you to generate random node values over the selected waveform, waveform interval, or across one or more nodes or groups. Figure 1–9 shows the **Random Values** dialog box.

You can generate random node values by every grid interval, every half grid interval, at random intervals, or at fixed intervals.

Figure 1–9. Random Values Dialog Box



Generating a Testbench

You can export your VWF as a VHDL Test Bench File (**.vht**) or Verilog Test Bench File (**.vt**). This is useful when you want to use a vector waveform in different EDA tools. You must run an analysis and

elaboration before you can export a waveform vector. To export a waveform vector, have your vector waveform open and perform the following steps:

1. On the File menu, click **Export**. The **Export** dialog box appears.
2. In the **Save as type** list, select **VHDL Test Bench File (*.vht)** or **Verilog Test Bench File (*.vt)**.
3. You can optionally turn on **Add self-checking code to file**. This option adds additional logic to check the results of the output and compares it to the original VWF.

You must open your project in the Quartus II software before you can export a VWF.

For more information about using the generated test bench in other EDA tools, refer to the respective EDA simulator chapter in the *Simulation* section in volume 3 of the *Quartus II Handbook*.

Grid Size

When you select portions of your waveform, the selection area snaps to time intervals specified in the **Grid Size** dialog box. You can customize the grid size in the Waveform Editor. You can change the grid size based on the clock settings or by setting the time period. To customize the grid size, on the Edit menu, click **Grid Size**.

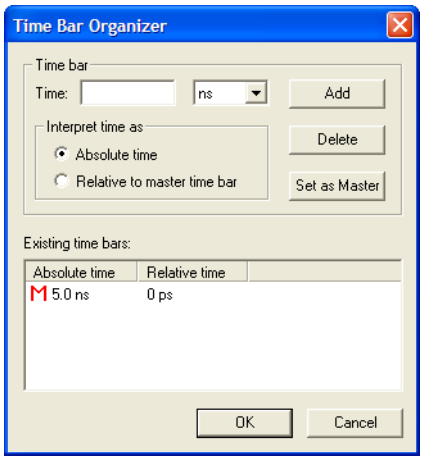
Time Bars

Add time bars in the Waveform Editor to compare edges between different signals. You can also use time bars to jump forward and backward to the next edge transition in the selected signal, and read the logic level of signals by sliding the Time Bar in your waveform. The logic level is displayed in the **Value at** column of the Waveform Editor.

The **Time Bar Organizer** dialog box enables you to create, delete, and edit a time bar, and to create a master time bar. Only one master time bar is allowed per waveform file. To use the Time Bar Organizer, on the Edit menu, point to Time Bar and click **Time Bar Organizer**.

Under **Existing time bars**, in the **Absolute time** column, the red **M** indicates the master time bar (Figure 1–10).

Figure 1–10. Time Bar Organizer Dialog Box



Stretch or Compress a Waveform Interval

You can stretch or compress a waveform interval in the Waveform Editor, which enables you to analyze the effects on a waveform. For example, you can check the behavior of your design at high speeds for a short interval by using the compress option to compress the waveform. You can also use this feature to delay the transition of a signal by stretching the waveform.

You have to specify the original start and end time, and the new time for the waveform you want to stretch or compress. If you want to stretch or compress all the nodes or buses, deselect all nodes and buses and set the stretch or compress feature.

To stretch or compress a waveform interval, on the Edit menu, point to Value and click **Stretch or Compress Waveform Interval**. The **Stretch or Compress Waveform Interval** dialog box appears.

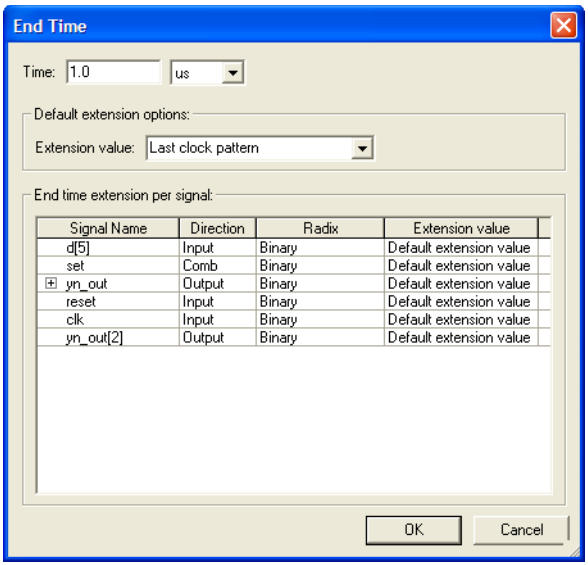
The "To time value" end time specified in the **Stretch or Compress Waveform Interval** dialog box cannot be larger than the "End Time" specified in the **Simulator Settings** page of the **Settings** dialog box (Figure 1–12). Otherwise, the Quartus II software displays a message indicating the invalid time value.

End Time

The End Time setting enables you to change the end time of the VWF. The end time represents the maximum length of time in the VWF. You can specify the end time and your preferred time unit, and have different extension values for different nodes or buses. With the waveform open, specify the end time by performing the following steps:

1. On the Edit menu, click **End Time**. The **End Time** dialog box appears (Figure 1–11).

Figure 1–11. End Time Dialog Box



2. In the **Time** box, specify the end time and select the time unit in the **Time** list.

3. Under **Default extension options**, in the **Extension value** list, select the value.

4. Under **End time extension per signal**, you can select specific extension values for each signal by clicking in the **Extension value** column.



The options in the **End time** dialog box are different settings than those under **Simulation period** in the **Settings** dialog box. Simulation period is the period that the Quartus II software simulates the stimuli. End time is the maximum length of time in the VWF. For information on the simulation period, refer to Table 1–2 on page 1–19.

Arrange Group or Bus in LSB or MSB Order

You can arrange a group or bus in Least Significant Bit (LSB) or Most Significant Bit (MSB) order. If you arrange in LSB order, the LSB is on top and MSB is at bottom. If you arrange in MSB order, the MSB is on top and LSB is at bottom.

To arrange a group or bus in LSB or MSB order, perform the following steps:

1. Select the bus that you want to change the LSB or MSB order. You can also select multiple buses in the waveform editor.
2. On the Edit, point to Group and Bus Bit Order and click either **MSB on top, LSB on Bottom** to change the bus or group in MSB order or click **LSB on top, MSB on Bottom** to change the bus or group in LSB order.

Simulator Settings

You can enhance your output, reduce debugging time, and provide better coverage before running a simulation. This section covers the different simulation modes supported by the Quartus II Simulator. Additionally, the Quartus II Simulator offers common setup features like glitch filtering, setup and hold violation detection, and simulation coverage.

To setup simulation settings, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulator Settings**. The **Simulator Settings** page appears (Figure 1–12).

Figure 1–12. Simulator Settings Page

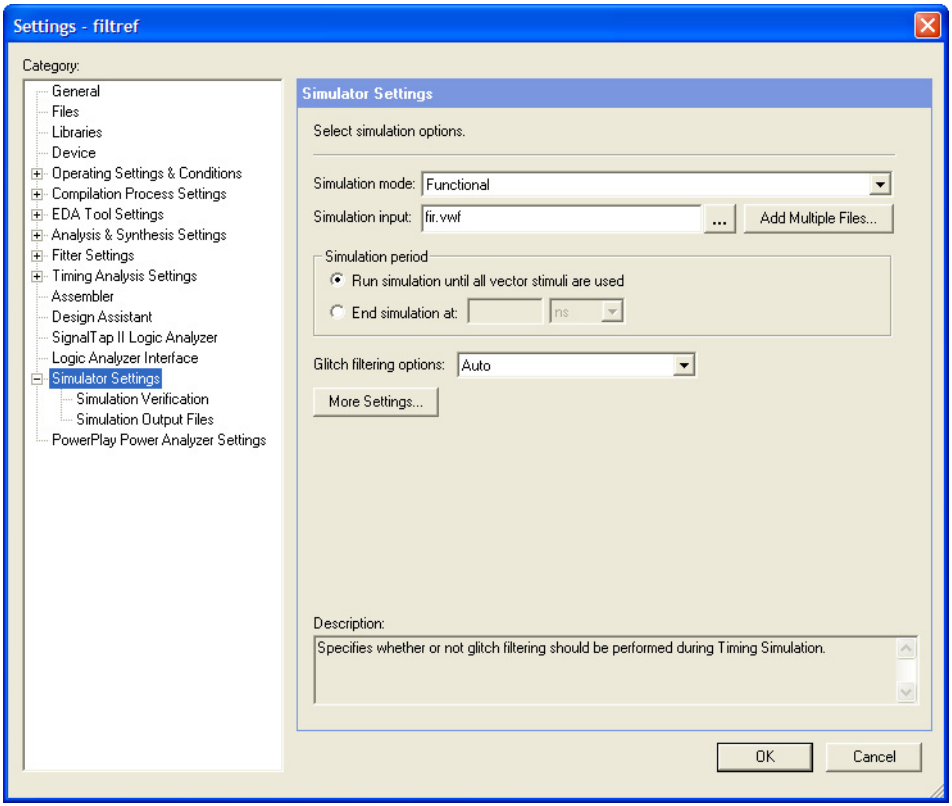


Table 1–2 shows the options in the **Simulator Settings** page.

Table 1–2. Quartus II Simulator Settings (Part 1 of 2)

Settings and Options	Description
Simulation mode *(1)*	**Functional** This simulation mode uses a pre-synthesis compiler database to simulate the logical performance of a project without the timing information. This mode enables you to check the functionality of the design. All nodes and buses are preserved in this simulation because functional simulation is performed before synthesis, partitioning, or fitting. A VWF is required to perform this simulation mode. **Timing** This simulation mode takes the compiled netlist that includes timing information. With this simulation mode, you can check setup, hold violation, glitches, and simulation coverage. You can remove nodes or buses using the Quartus II Compiler when logic is optimized. This simulation mode uses the worst case timing model. **Timing using Fast Timing Model** This simulation mode is similar to timing simulation but this mode uses the best-case timing model.
Simulation input	You must include the vector file in the **Simulation input** box. You can type the name of the file or use the browse button to open the **Select File** dialog box. In the **Files of type** list, you can select Vector Waveform File (***.vwf**), Compressed Vector Waveform File (***.cvwf**), Value Change Dump File (***.vcd**), Vector Table Output File (***.tbl**), Vector Text File (***.vec**), Simulation Channel File (**.scf**), or **All Files (*.*)**. TBL files contain input vectors and output logic levels in tabular-format list. You can generate this file using a VWF. However, if you would like to maintain, view, or update the vectors, VWFs offer better visibility. VWF or TBL file formats are interchangeable. You can generate TBL files from VWFs and vice versa. You can create a VWF with the Waveform Editor. For more information on the Waveform Editor, refer to "Waveform Editor" on page 1–5. The Quartus II software also supports MAX+PLUS® II simulation vector files, such as VEC and SCF. A CVWF is the simplified version, non-readable, format of the VWF format. This file type is in binary format and is generally smaller in file size. You can use CVWFs in the Waveform Editor and simulation. A VCD file is an ASCII file which contains header information, variable definitions, and the value changes for specified variables, or all variables, in a given design. The value changes for a variable are given in scalar or vector format, based on the nature of the variable.

Table 1–2. Quartus II Simulator Settings (Part 2 of 2)

Settings and Options	Description
Simulation period	The simulation period determines the length of time that the simulator runs the stimuli with the maximum period being equal to the end time of a VWF. If the simulation period is configured shorter than the end time, all signals beyond the simulation period are displayed as Unknown (X). Therefore, you can also shorten the simulation period or end the simulation earlier by selecting **End Simulation at** and specifying the time and selecting the time unit. If the simulation period is configured longer than the end time, the simulation will stop at the end time. For information on the end time, refer to "End Time" on page 1–16.
Glitch filtering options	Specifies whether to enable glitch filtering for simulations. You can select one of the following options: **Auto**—The Simulator performs glitch filtering when SAF generation is enabled in the **Simulation Output Files** page of the **Settings** dialog box. **Always**—The Simulator always performs glitch filtering, even if SAF generation is not enabled. **Never**—The Simulator never performs glitch filtering, even if SAF generation is enabled.
More Settings	If you click **More Settings**, the **More Simulator Settings** dialog box appears. The following options are available under **Existing option settings**. **Cell Delay Model Type** Specifies the type of delay model to be used for cell delays: transport or inertial. The default is transport. **Interconnect Delay Model Type** Specifies the type of delay model to be used for interconnect delays: transport or inertial. The default is transport. **Preserve fewer signal transition to reduce memory requirements** This option is effective on lower performance workstations because turning on this option flushes signal transitions from memory to disk for memory optimization.

Note to Table 1–2:

(1) The Quartus II Simulator may flag an error message if zero-time oscillation happens in your design. Zero-time oscillation happens when a particular output signal does not achieve a stable output value at a particular fixed time, which may be due to your design containing combinational logic path loops.

Simulation Verification Options

Figure 1–13 shows the simulation verification page.

Figure 1–13. Simulation Verification Page

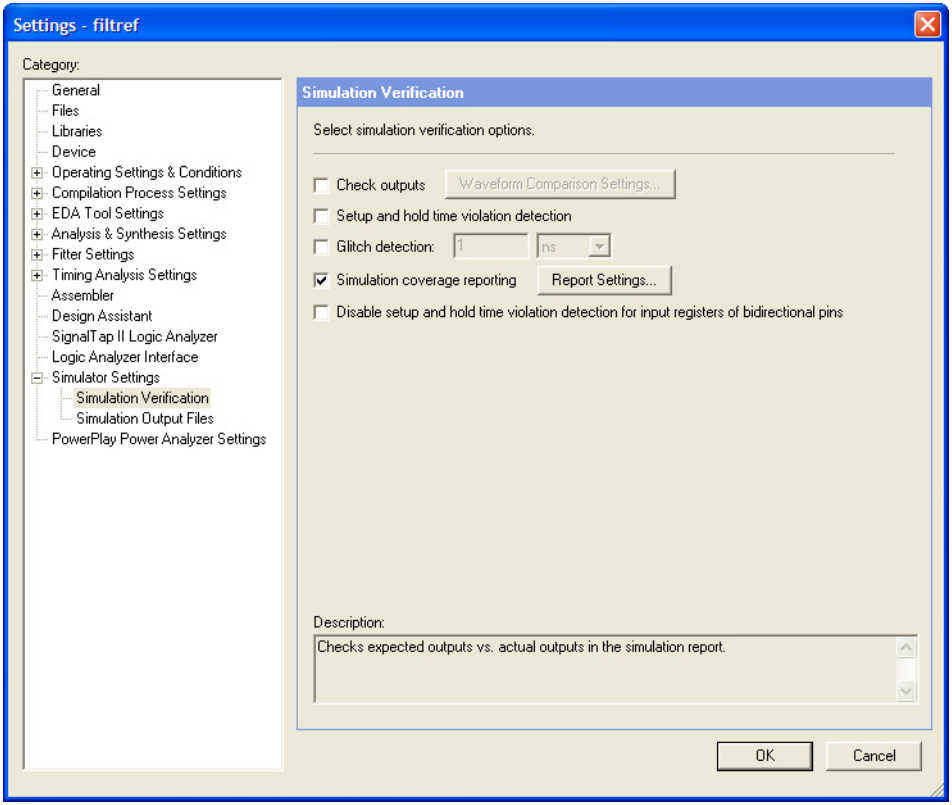


Table 1–3 shows the options in the simulation verification page.

Table 1–3. Quartus II Simulation Verification (Part 1 of 2)

Settings and Options	Description
Check outputs	**Check outputs** checks expected outputs against actual outputs in the simulation report. After turning on **Check outputs**, click the **Waveform Comparison Settings** button. The **Waveform Comparison Settings** dialog box appears. In the **Waveform Comparison Settings** dialog box, you can specify the waveform comparison time frame and the comparison options. You can also set the tolerance level for all the signals by specifying the tolerance limit in the **Default comparison timing tolerance** box. The **Maximum comparison mismatches** box is the amount of mismatches the Quartus II Simulator is allowed to accept before it stops comparing. You can also set the type of transition the comparison should trigger in the **Waveform Comparison Settings** dialog box. You can assign trigger comparisons based on **Input signal transition edges**, **All signal transition edges**, or **Selected Signal transition edges**. To customize the waveform comparison matching rules, you can also click the **Comparison Rules** button. The **Comparison Rules** dialog box appears, allowing you to customize the comparison matching rules.
Setup and hold time violation detection	This option detects setup and hold time violation. Setup time is the period required by a synchronous signal to stabilize before the arrival of a clock edge. Hold time is the time required by a synchronous signal to maintain after the same clock edge. If the **Setup and hold time violation detection** option is turned on, a warning in the Messages windows appears if any setup or hold time violation is detected during the simulation. This option is only for Timing and Timing using Fast Timing Model simulation modes.
Glitch detection	Conditions happen when two or more signals toggle simultaneously and can cause glitches or unwanted short pulses. The **Glitch detection** option enables you to detect glitches and specify the time interval that defines a glitch. If two logic level transitions occur in a period shorter than the specified time period, the resulting glitch is detected and reported in the **Processing** tab of the Messages window. If you turn on the **Glitch detection** option, you can specify the acceptable glitch width. A Messages window appears when a pulse is smaller than the specified glitch width that is detected. The **Glitch detection** option is only available for Timing and Timing using Fast Timing Model simulation modes.

Table 1–3. Quartus II Simulation Verification (Part 2 of 2)

Settings and Options	Description
Simulation coverage reporting	This option reports the ratio of outputs (coverage) actually simulated to the number of outputs in the netlist and is expressed as a percentage. When you turn on the **Simulation coverage reporting** option, the **Report Settings** button is available. If you click **Report Settings**, the **Report Settings** dialog box appears. The three types of coverage reports you can select from are **Display complete 1/0 value coverage report**, **Display missing 1-value coverage report**, and **Display missing 0-value coverage report**.
Disable setup and hold time violation detection for input registers of bi-directional pins	This option enables you to disable setup and hold time violations detection in input registers of all bidirectional pins in the simulated design during Timing or Timing using Fast Timing Model simulation.

Simulation Output Files Options

Figure 1–14 shows the simulation output file page.

Figure 1–14. Simulation Output Files Page

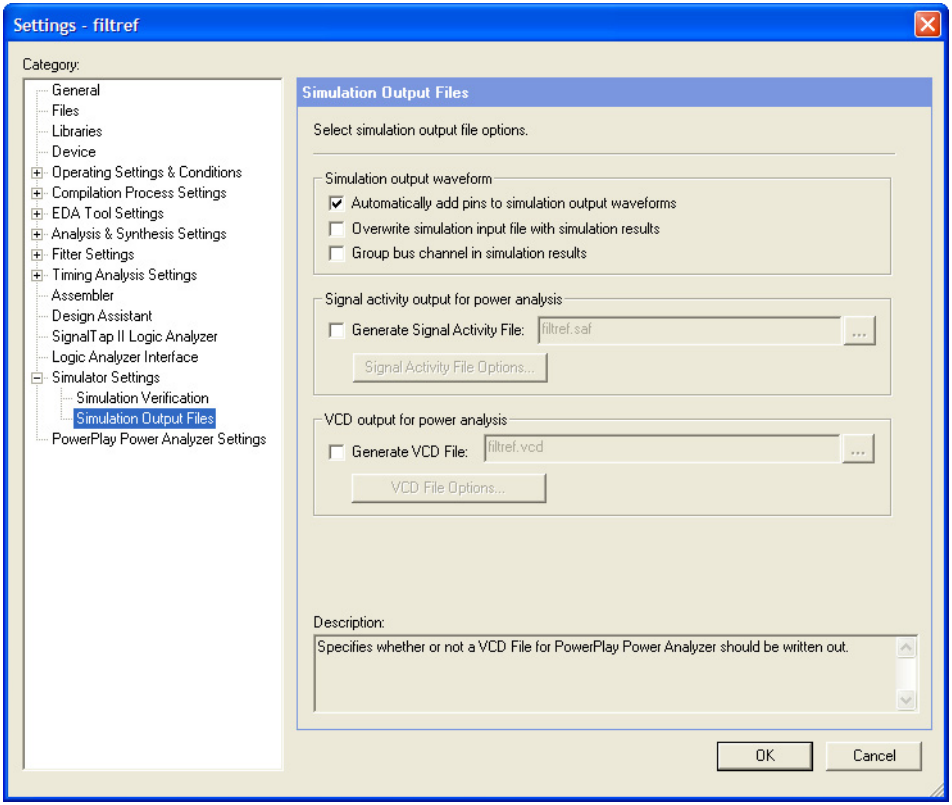


Table 1–4 shows the options in the simulation output file page.

Table 1–4. Quartus II Simulation Output Files

Setting and Options	Description
Simulation output waveform	Specify the simulation output waveform options. **Automatically add pins to simulation output waveforms** The **Automatically add pins to simulation output waveforms** option automatically adds all outputs that are available in the design to the waveform reports. If your design has large amounts of outputs, turning on this option ensures all outputs are monitored during simulation. **Overwrite simulation input file with simulation results** This option overwrites the vector source file with simulation results. This option is ignored when the **Check outputs** setting is turned on. This option adds the result to the vector file and generally, it can give you more visibility during the debugging process. *(1)* **Group bus channel in simulation results** This option automatically groups bus channels in the output waveform that are shown in the simulation reports. By turning off this option, all output waveforms have a node to represent each bus signal.
Signal activity output for power analysis	When you perform your simulation with the Quartus II Simulator, you can generate a SAF which is used by the PowerPlay Power Analyzer to assist you with power analysis. *(2)*, *(3)*
VCD output for power analysis	When you perform simulation with the Quartus II Simulator, you can generate a VCD file, which is used by the PowerPlay Power Analyzer to assist you with power analysis. *(2)*, *(3)*

Notes to Table 1–4:

(1) A backup copy of the source vector file is saved under the **db** folder with the name <*project*>**.sim_ori.** <*vector file format type*>.

(2) Instead of using the SAF or Generate VCD file (***.vcd**), you can also save your output waveform as a VCD file to perform power analysis.

(3) For more information about the PowerPlay Power Analyzer, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Simulation Report

Comprehensive reports are shown after the completion of each simulation. These reports are important to ensure designs meet timing and logical correctness. These simulation reports also play an important role in debugging.

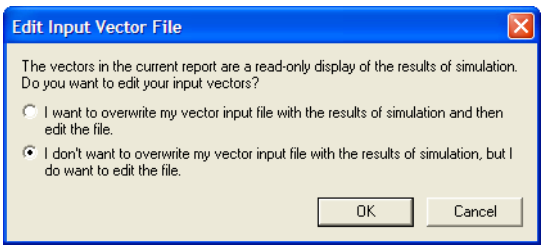
Simulation Waveform

Simulation Waveforms are part of the Simulation report. In this report, the stimuli and the results of the simulation are displayed.

You can export the simulation waveform as a VHDL Test Bench File or a Verilog Test Bench File for use in other EDA tools. You can also save a simulation as a VWF or Vector Table Output File for use with the Quartus II software.

When you try to edit the Simulation Waveform, the **Edit Input Vector File** dialog box appears, asking whether you would like to edit the vector input file with the results of the simulation or if you would like to overwrite the vector input file with other vector inputs (Figure 1–15).

Figure 1–15. Edit Input Vector File



You can overwrite your simulation input file with the simulation results so that your input vector file is updated with the resulting waveform after a simulation. For more information, refer to the **Overwrite simulation input file with simulation results** option in Table 1–2.

If you do not want to overwrite the simulation input file in every simulation run, perform the following to overwrite simulation input files with simulation results after a simulation:

On the Processing Menu, point to Simulation Debug and click **Overwrite Vector Inputs with Simulation Outputs**.

Simulating Bidirectional Pin

A bidirectional pin is represented in the waveform by two channels. One channel represents the input to the bidirectional pin, and the other channel represents the output from the bidirectional pin. You can enter the input channel into the waveform by using the **Node Finder** dialog box. The output channel is automatically created by the Quartus II Simulator and named <*bidir pin name*> `~result`.

Logical Memories Report

The Quartus II software writes out the contents of each memory module after simulation. Therefore, if you use memory cells in your design, you can analyze the contents of the logic memory structures in the device in the Logical Memories Report. The Logical Memories Report displays individual reports for each memory block and contains the data stored in the memory cell used at the end of simulation.

After being simulated, a memory module's contents are stored in the Logical Memories section of the simulation report file.

To view this section, perform the following steps:

1. On the Processing menu, click **Simulation Report**. The Simulation Report window appears.

2. In the report window, click on the "+" next to **Logical Memories**.

Simulation Coverage Reports

The **Coverage Summary** report contains the following summary information for the simulation:

- Total toggling coverage as a percentage
- Total nodes checked in the design
- Total output ports checked
- Total output ports with complete 1/0-value coverage
- Total output ports with no 1/0-value coverage
- Total output ports with no 1-value coverage
- Total output ports with no 0-value coverage

The **Complete 1/0-Value Coverage** report lists the following information:

- Node name
- Output port name
- Output port type for output ports that toggle between 1 and 0 during the simulation

The **Missing 0-Value Coverage** report and **Missing 1-Value Coverage** report list the following information:

- Node name
- Output port name
- Output port type for output ports that do not toggle to the designated value

For more information about Simulation Coverage reports, refer to the **Simulation coverage reporting** option in Table 1–2 on page 1–19.

The following are individual reports and their definition:

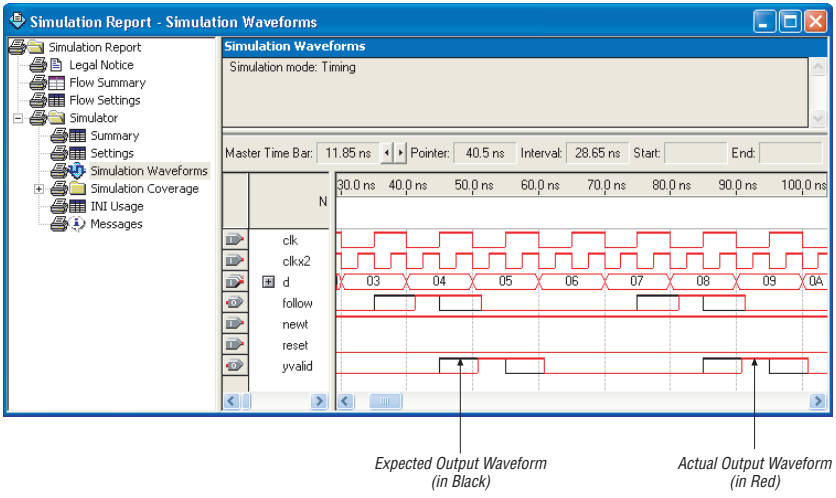
Complete 1/0 value coverage report
Displays all the nodes or buses that toggle between 1 and 0 during simulation.

Missing 1-value coverage and **Missing 0-value coverage reports**
Displays all the nodes that do not toggle to the designated value.

Comparing Two Waveforms

You can compare your simulation results against previous simulations using the compare option. To compare two waveforms in the **Simulation Report**, turn on the **Check outputs** option. For more information on the **Check outputs** option, refer to Table 1–2 on page 1–19. With the **Check outputs** option turned on, the two comparable waveforms are visible in black and red. The black waveforms represent the original output or the expected output, and the red waveforms represent the compared output or the actual output. Figure 1–16 shows an example of expected output waveform versus actual output waveform.

Figure 1–16. Example of Simulation Waveform from the Simulation Report When Check Output is Turned On



Debugging with the Quartus II Simulator

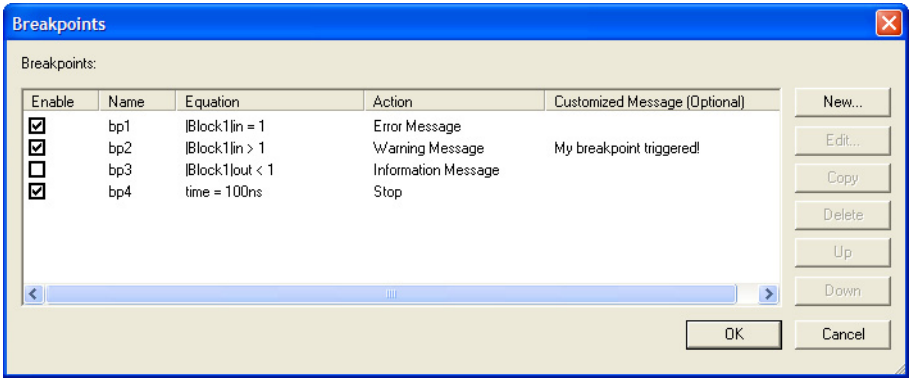
The Quartus II software includes tools to help with simulation debugging. This section covers some debugging tools and their use.

Breakpoints

Inserting breakpoints into the simulation process enables the simulator to break at the desired time or on the desired node or bus condition. You can monitor the activity of nodes or buses during specified times and pinpoint the cause of mismatched signal levels between expected and actual. To use breakpoints, perform the following steps:

1. On the Processing menu, point to Simulation Debug and click **Breakpoints**. The **Breakpoints** dialog box appears (Figure 1–17).

Figure 1–17. Breakpoints Dialog Box



2. Click **New** to create a new breakpoint. The **New Breakpoint** dialog box appears. In this dialog box, you can specify the name, the equation, and the action of the breakpoint. You can also enable or disable this breakpoint by using the **Enable Breakpoint** check box.

3. In the Equation text box, click **condition**. You can configure the logical conditions of individual nodes or buses, or you can set the time.

4. After you configure the equation conditions, select the action for the Quartus II Simulator. In the Action drop down list, select **Stop**, **Warning Message**, **Error Message**, or **Information Message**. This selection defines the action once the condition is met.

5. You can also enter the text that appears when the Simulator encounters the breakpoint. If you do not make an entry in this box, the Quartus II software displays a default message.

Updating Memory Content

If your design includes memories, when the simulator stops at a breakpoint, you can view and edit the contents of the memories. To view your memories during a breakpoint in the simulation, on the Processing menu, point to Simulation Debug and click **Embedded Memory**.

Last Simulation Vector Outputs

The Last Simulation Vector Outputs command opens the Output Simulation Waveforms report generated by the last simulation. To use this command, on the Processing menu, point to Simulation Debug and click **Last Simulation Vector Outputs**.

You can open the current input vectors that you defined in the **Simulator Settings** dialog box with the Current Vector Inputs command. To use this command, on the Processing menu, point to Simulation Debug and click **Current Vector Inputs**. Lastly, you can overwrite the vector source file with the simulation outputs which open the resulting file.

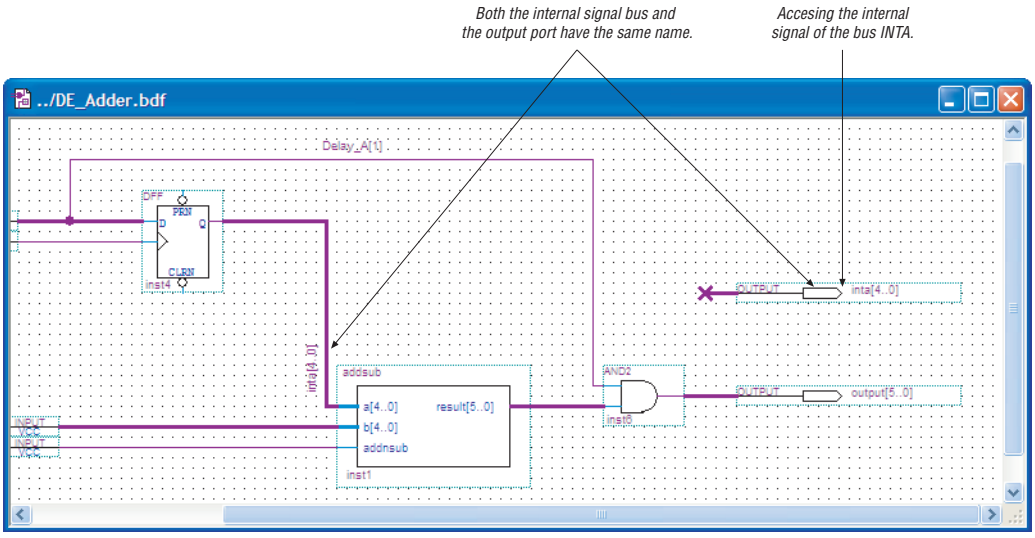
Conventional Debugging Process

During the design phase, tapping out internal signals is a common practice to debug simulation errors. Therefore, the Quartus II software enables you to tap out the signal for simulation debug and also enables you to pull out the internal signal to the physical I/O. The Quartus II software also offers SignalTap II and SignalProbe to further assist you with debugging.

Accessing Internal Signals for Simulation

You can conventionally debug by probing out the internal signals, which enables you to preserve the internal signals during synthesis. You can probe the internal signal by selecting the node or bus and specifying a name, and then adding an output port to the schematic with a similar name. Figure 1–18 shows an example of accessing internal signals for simulation from a schematic diagram.

Figure 1–18. Example of Tapping Out Internal Signal



For timing simulations, the simulation netlist is based on the Compilation post-Synthesis and post-Fitting netlist. Therefore, some of the internal nodes or buses are optimized away during compilation of the netlist. If an internal node is optimized away, the Quartus II software shows a warning in the **Warning** tab of the Messages window similar to the following message:

```
Warning: Compiler packed, optimized or synthesized away node "DataU". Ignored vector
source file node.
```

This internal node is ignored by the Quartus II Simulator.

If you would like to tap out the D and Q ports of registers, turn on **Add D and Q ports of register node to Simulation Output Waveform** from the Assignment Editor. This feature is only available for functional simulations.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The Scripting Reference Manual includes the same information in PDF form.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

You can change the Functional, Timing, or Timing using Fast Timing Model simulation modes with the following command:

```
simulation_mode <mode> ↵
```

To initialize the simulation for the current design, use the following command. During initialization, the Simulator builds the simulation netlist and sets the simulation time to zero.

The option `-ignore_vector_file` is set to **Off** by default, when the source vector file exists for simulation. The Quartus II software ignores the source vector file during simulation if the option `-ignore_vector_file` is set to On. The `-end_time` option is used only when the `-ignore_vector_file` option is set to **On**.

```
initialize_simulation [-h | -help] [-long_help] [-check_outputs <On | Off>] \
[-end_time <end_time>] [-glitch_filtering <On | Off>] [-ignore_vector_file <On | Off>] \
[-memory_limiter <On | Off>] [-power_vcd_output <target_file>] [-read_settings_files <On |
Off>] \ [-saf_output <target_file>] [-sim_mode <functional | timing | timing_using_fast_timing_model >] \
[-vector_source <vector_source_file>] [-write_settings_files <On | Off>] \
-simulation_results_format <VWF | CVWF | VCD> -vector_source <vector source file>
```

To force the specified signal or group of signals to the specified value, type the following at a command prompt:

```
force_simulation_value [-h | -help] [-long_help] -node <hpath> <value> ↵
```

To turn on the simulator to simulate the design for a specified time, type the following at a command prompt:

```
run_simulation [-h | -help] [-long_help] [-time <time>] ↵
```



If you do not specify the length of time the simulation runs, it runs until the simulation is complete.

To create a breakpoint with a specified equation and action, type the following at a command prompt:

```
create_simulation_breakpoint [-h | -help] [-long_help] \
-action [Give Warning | Give Info | Give Error] \
-breakpoint <breakpoint_name> -equation <equation> [-user_message <message_text>]↵
```

To delete a breakpoint with a specified name, type the following at a command prompt:

```
delete_simulation_breakpoint [-h | -help] [-long_help] \
-breakpoint <breakpoint_name>↵
```

Conclusion

Simulation plays an important role in ensuring the quality of a product. The Quartus II software offers various tools to assist you with simulation and helps reduce debugging time with the introduction of features like Glitch Filtering and Breakpoints.

Referenced Documents

This chapter references the following documents:

- *Quartus II Settings File Reference Manual*
- *Section I: Simulation* section in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 1–5 shows the revision history for this chapter.

Table 1–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 1–33.	—
May 2007 v7.1.0	• Updated a command in Scripting Support. • Updated Breakpoints. • Added procedure to Logical Memories Report. • Updated sections, added sections and deleted sections in Simulator Settings. • Updated Simulation Report. • Updated Table 1-2. • Added Arrange Group or Bus in LSB ir MSB Order. • Updated Creating VWFs. • Added Referenced Documents.	Updated for the Quartus II software version 7.1.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Updated for the Quartus II software version 6.1. • Added references to Value Change Dump File (**.vcd**) • Added Random Value section • Other minor changes	Updated for the Quartus II software version 6.1.
May 2006 v6.0.0	Initial release.	—

2. Mentor Graphics ModelSim Support

QII53001-7.2.0

Introduction

An Altera® software subscription includes a license for the ModelSim-Altera software on a PC or UNIX platform. The ModelSim-Altera software can be used to perform functional register transfer level (RTL), post-synthesis, and gate-level timing simulations for either Verilog HDL or VHDL designs that target an Altera FPGA. This chapter provides detailed instructions on how to simulate your design in the ModelSim-Altera version or the Mentor Graphics® ModelSim® software version. This chapter gives you details on the specific libraries that are needed for a functional RTL simulation or a gate-level timing simulation.

This document describes using ModelSim-Altera software version 6.1g and the Mentor Graphics ModelSim software version 6.1g. It also contains references to features available in the Altera Quartus® II software version 7.2.

The following topics are discussed in this chapter:

- "Background"
- "Software Compatibility" on page 2–3
- "Altera Design Flow with ModelSim or ModelSim-Altera Software" on page 2–3
- "Functional RTL Simulation" on page 2–5
- "Post-Synthesis Simulation" on page 2–16
- "Gate-Level Timing Simulation" on page 2–23
- "Simulating Designs that Include Transceivers" on page 2–37
- "Using the NativeLink Feature with ModelSim" on page 2–44
- "Scripting Support" on page 2–50
- "Software Licensing and Licensing Setup" on page 2–51



For more information about the current Quartus II software version, refer to the Altera website at www.altera.com.

Background

The ModelSim-Altera software version 6.1g is included with your Altera software subscription and can be licensed for the PC, Solaris, or Linux platforms to support either Verilog HDL or VHDL hardware description language (HDL) simulation. The ModelSim-Altera software supports VHDL or Verilog functional RTL, post-synthesis, and gate-level timing simulations for all Altera devices.

Table 2–1 describes the differences between the Mentor Graphics ModelSim SE/PE and ModelSim-Altera software versions.

Table 2–1. Comparison of ModelSim Software Versions

Product Feature	ModelSim SE	ModelSim PE	ModelSim-Altera	ModelSim-Altera Web Edition
100% VHDL, Verilog, mixed-HDL support	Optional	Optional	Supports only single-HDL simulation	Supports only single-HDL simulation
Complete HDL debugging environment	✓	✓	✓	✓
Optimized direct compile architecture	✓	✓	✓	✓
Industry-standard scripting	✓	✓	✓	✓
Flexible licensing	✓	Optional	✓	—
Verilog PLI support. Interfaces Verilog HDL designs to customer C code and third-party software	✓	✓	✓	✓
VHDL FLI support. Interfaces VHDL designs to customer C code and third-party software	✓	—	—	—
Standard Delay Format File annotation	✓	✓	✓ *(1)*	✓ *(1)*
Advanced debugging features and language-neutral licensing	✓	—	—	—
Customizable, user-expandable graphical user interface GUI and integrated simulation performance analyzer	✓	—	—	—
Integrated code coverage analysis and SWIFT support	✓	—	—	—
Accelerated VITAL and Verilog HDL primitives (3 times faster), and register transfer level (RTL) acceleration (5 times faster)	✓	—	—	—
Platform support	PC, UNIX, Linux	PC only	PC, UNIX, Linux	PC only
Precompiled Libraries	No	No	Yes	Yes

Note to Table 2–1:
(1) ModelSim-Altera will only allow SDF annotation to modules in the Altera library.

Software Compatibility

Table 2–2 shows which ModelSim-Altera software version is compatible with the Quartus II software versions. ModelSim versions provided directly from Mentor Graphics do not correspond to specific Quartus II software versions.

For help with ModelSim-Altera licensing set up, refer to "Software Licensing and Licensing Setup" on page 2–51.

Table 2–2. Compatibility Between Software Versions

ModelSim-Altera Software	Quartus II Software *(1)*
ModelSim-Altera software version 6.1g	Quartus II software version 6.1, 7.0, 7.1, and 7.2
ModelSim-Altera software version 6.1d	Quartus II software version 6.0
ModelSim-Altera software version 6.0e	Quartus II software version 5.1
ModelSim-Altera software version 6.0c	Quartus II software version 5.0
ModelSim-Altera software version 5.8.e ModelSim-Altera software version 5.8	Quartus II software version 4.2

Note to Table 2–2:
(1) Updated ModelSim-Altera precompiled libraries are available for download on Altera's website for each release of the Quartus II service pack.

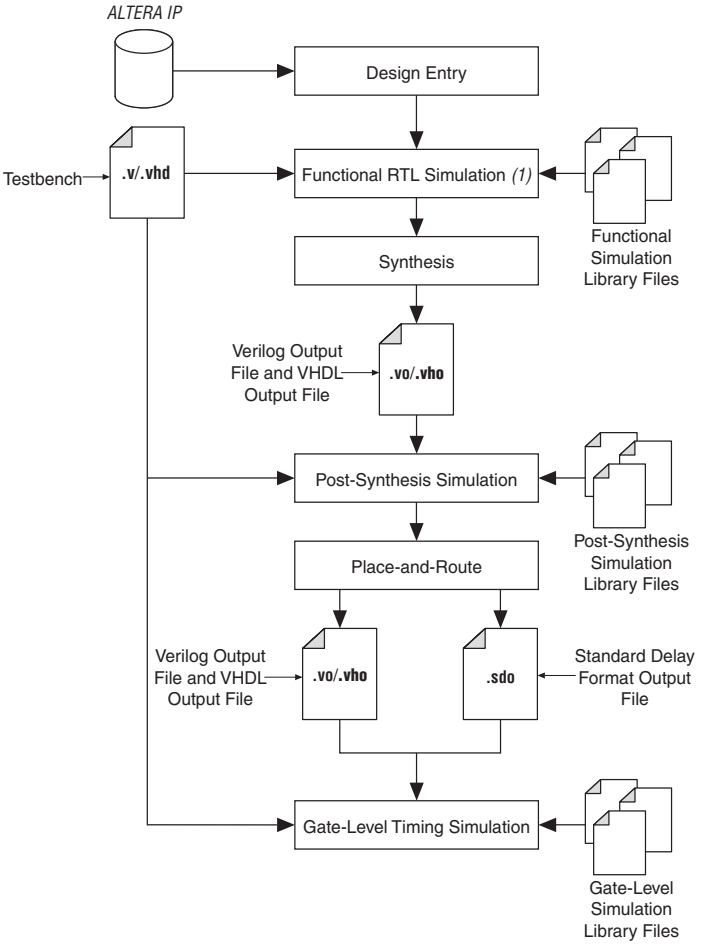
Altera Design Flow with ModelSim or ModelSim-Altera Software

This chapter contains the following sections:

- Functional RTL simulations
- Post-synthesis simulations
- Gate-level timing simulations
- Using the NativeLink® feature with ModelSim

Figure 2–1 illustrates an Altera design flow using the Mentor Graphics ModelSim software or ModelSim-Altera software.

Figure 2–1. Altera Design Flow with ModelSim-Altera and Quartus II Software



Note to Figure 2–1:

(1) If you are performing a functional simulation through NativeLink, you must complete analysis and elaboration first.

Functional RTL Simulation

A functional RTL simulation is performed before a gate-level simulation or post-synthesis simulation. Functional RTL simulation verifies the functionality of the design before synthesis and place-and-route. This section provides detailed instructions on how to perform a functional RTL simulation in the ModelSim-Altera software and highlights some of the differences in performing similar steps in the Mentor Graphics ModelSim software versions for Verilog HDL and VHDL designs.

Functional Simulation Libraries

Pre-compiled libraries are available for functional simulation with the ModelSim-Altera software. These libraries include the **lpm** library and the **altera_mf** library. To create these libraries for simulation with the ModelSim SE/PE software, compile the library files described in the following sections.

lpm Simulation Models

To simulate designs containing lpm functions, use the following functional simulation models:

- **220model.v** (for Verilog HDL)
- **220pack.vhd** and **220model.vhd** (for VHDL)



When you are simulating a design that uses VHDL-1987, use the **220model_87.vhd** model file.

Table 2–3 shows the location of these simulation model files and precompiled libraries in the Quartus II software and the ModelSim-Altera software.

Table 2–3. Location of lpm Simulation Models Files and Pre-Compiled Libraries

Software	Location
Quartus II	<*Quartus II installation directory*>**\eda\sim_lib** *(1)*
ModelSim-Altera	<*ModelSim-Altera installation directory*>**\altera\<HDL>\220model** *(2)*, *(3)*

Notes to Table 2–3:
(1) For ModelSim SE/PE, compile the files provided with the Quartus II software.
(2) For ModelSim-Altera, use the precompiled libraries for simulation.
(3) <*HDL*> can be either Verilog HDL or VHDL.



For more information about LPM functions, refer to the Quartus II Help.

Altera Megafunction Simulation Models

To simulate a design that contains Altera megafunctions, use the following simulation models:

- **altera_mf.v** (for Verilog HDL)
- **altera_mf.vhd** and **altera_mf_components.vhd** (for VHDL)



When you are simulating a design that uses VHDL-1987, use **altera_mf_87.vhd.**

Table 2–4 shows the location of these simulation files and precompiled libraries in the Quartus II software and the ModelSim-Altera software.

Table 2–4. Location of Altera Megafunction Simulation Models Files and Precompiled Libraries

Software	Location
Quartus II	*<Quartus II installation directory>***\eda\sim_lib** *(1)*
ModelSim-Altera	*<ModelSim-Altera installation directory>***\altera\<*HDL*>\altera_mf** *(2)*, *(3)*

Notes to Table 2–4:
(1) For ModelSim SE/PE, compile the files provided with the Quartus II software.
(2) For ModelSim-Altera, use the precompiled libraries for simulation.
(3) *<HDL>* can be either Verilog HDL or VHDL.

The following Altera megafunctions require device atom libraries to perform a functional simulation in a third-party simulator:

- altclkbuf
- altclkctrl
- altdqs
- altdq
- altddio_in
- altddio_out
- altddio_bidir
- altufm_none
- altufm_parallel
- altufm_spi
- altmemmult
- altremote_update

The device atom library files are located in the following directory:

*<Quartus II installation directory>***/eda/sim_lib**

Low-Level Primitive Simulation Models

You can simulate a design that contains low-level Altera primitives with the following simulation models:

- **altera_primitives.v** (for Verilog HDL)
- **altera_primitives.vhd** and **altera_primitives_components.vhd** (for VHDL)

Table 2–5 shows the location of these simulation library files and precompiled libraries in the Quartus II software and the ModelSim-Altera software.

Table 2–5. Location of Altera Primitives Model Files and Precompiled Libraries

Software	Location
Quartus II	<*Quartus II installation directory*>\eda\sim_lib *(1)*
ModelSim-Altera	<*ModelSim-Altera installation directory*>\altera\<HDL>\altera *(2)*, *(3)*

Notes to Table 2–5:
(1) For ModelSim SE/PE, compile the files provided with the Quartus II software.
(2) For ModelSim-Altera, use the precompiled libraries for simulation.
(3) <*HDL*> can be either Verilog HDL or VHDL.

Simulating VHDL Designs

Use the following instructions to perform a functional RTL simulation for VHDL designs in the ModelSim software.



The steps in the following section assume you have already created a ModelSim project.

The ModelSim-Altera software comes with precompiled simulation libraries. Creating simulation libraries and compiling simulation models steps are not required. You can proceed directly to "Compile Testbench and Design Files into Work Library" on page 2–9.

Create Simulation Libraries

Simulation libraries are required to simulate a design that contains an Altera primitive, lpm function, or Altera megafunction. These libraries have already been compiled if you are using the ModelSim-Altera software. However, if you are using the Mentor Graphics ModelSim software, you must create the simulation libraries and link them to your design correctly.

Create Simulation Libraries Using the ModelSim GUI
Perform the following steps to create simulation libraries:

1. In the ModelSim software, on the File menu, point to New and click **Library**. The **Create a New Library** dialog box appears.
2. Select **a new library and a logical mapping to it**.
3. In the **Library Name** box, type the name of the newly created library.

   For example, the library name for Altera megafunctions should be **altera_mf**, and the library name for LPM should be **lpm**.
4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
vlib altera_mf ↵
vmap altera_mf altera_mf ↵
vlib lpm ↵
vmap lpm lpm ↵
vlib altera ↵
vmap altera altera ↵
```

Compile Simulation Models into Simulation Libraries

The following steps are not required for the ModelSim-Altera software.

Compile Simulation Models into Simulation Libraries Using the ModelSim GUI
Perform the following steps to compile simulation models into simulation libraries:

1. On the File menu, point to Add to Project and click **Existing File**.
2. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files to your project.

   The **altera_mf.vhd** model file should be compiled into the **altera_mf** library. The **220pack.vhd** and **220model.vhd** model files should be compiled into the **lpm** library.
3. In the Workspace window, select the simulation model file, and on the View menu, click **Properties**.

4. Choose the correct library from the **Compile to Library** list.
5. Click **OK**.
6. On the Compile menu, click **Compile selected**.

Compile Simulation Models into Simulation Libraries at the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
vcom -work altera_mf <Quartus II installation directory>/eda/sim_lib/altera_mf_components.vhd ↵
vcom -work altera_mf <Quartus II installation directory>/eda/sim_lib/altera_mf.vhd ↵
vcom -work lpm <Quartus II installation directory>/eda/sim_lib/220pack.vhd ↵
vcom -work lpm <Quartus II installation directory>/eda/sim_lib/220model.vhd ↵
vcom -work altera <Quartus II installation directory>/eda/sim_lib/altera_primitives_components.vhd ↵
vcom -work altera <Quartus II installation directory>/eda/sim_lib/altera_primitives.vhd ↵
```

Compile Testbench and Design Files into Work Library

Compile a testbench and design files into a work library by clicking **Compile All** or by clicking the **Compile All** toolbar icon on the Compile menu.

Compile Testbench and Design Files into Work Library Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vcom -work work <my_test bench.vhd> <my_design_files.vhd>↵
```



Resolve compile-time errors before proceeding to the following section.

Loading the Design

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. Expand the work library in the **Start Simulation** dialog box.
3. Select the top-level design unit (your testbench).
4. In the **Resolution** list, select **ps**.
5. Click **OK**.

Loading the Design Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vsim work.<my_test bench> -t ps ↵
```

Running the Simulation

Perform the following steps to run a simulation:

1. On the View menu, point to Debug Windows and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to Debug Windows and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. Type the following command at the ModelSim command prompt:

   `run` <*time period*> ↵

Running the Simulation Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
add wave /<signal name> ↵
run <time period> ↵
```

Simulating Verilog HDL Designs

The following instructions provide step-by-step instructions to perform functional RTL simulation for Verilog HDL designs in the ModelSim software.



The following steps assume you have already created a ModelSim project.

If you are using the ModelSim-Altera software, a set of precompiled libraries are created when you install the software. Creating simulation libraries and compiling simulation models steps are not required. You can proceed directly to "Compile Testbench and Design Files into Work Library" on page 2–12.

Create Simulation Libraries

Simulation libraries are needed to properly simulate a design that contains an lpm function or an Altera megafunction. These libraries have already been compiled if you are using the ModelSim-Altera software.

However, if you are using the Mentor Graphics ModelSim software, you must create the simulation libraries and correctly link them to your design.

Create Simulation Libraries Using the ModelSim GUI
Perform the following steps to create simulation libraries:

1. On the File menu, point to New and click **Library**. The **Create a New Library** dialog box appears.
2. Select **a new library and a logical mapping to it**.
3. In the **Library Name** box, type the name of the newly created library.

   For example, the library name for Altera megafunctions should be **altera_mf**, and the library name for LPM should be **lpm**.
4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
vlib altera_mf ↵
vmap altera_mf altera_mf ↵
vlib lpm ↵
vmap lpm lpm ↵
vlib altera ↵
vmap altera altera ↵
```

Compile Simulation Models into Simulation Libraries

The following steps are not required for the ModelSim-Altera software.

Compile Simulation Models into Simulation Libraries Using the ModelSim GUI
Perform the following steps to compile simulation models into simulation libraries:

1. On the File menu, point to Add to Project and click **Existing File**.
2. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files to your project.



Compile the **altera_mf.v** into the **altera_mf** library. Compile the **220model.v** into the **lpm** library.

3. Select the simulation model file and on the View menu, click **Properties**.
4. Choose the correct library from the **Compile to Library** list.
5. Click **OK**.
6. On the Compile menu, click **Compile selected**.

Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
vlog -work altera_mf <Quartus II installation directory>/eda/sim_lib/altera_mf.v ↵
vlog -work lpm <Quartus II installation directory>/eda/sim_lib/220model.v ↵
vlog -work altera <Quartus II installation directory>/eda/sim_lib/altera_primitives.v ↵
```

Compile Testbench and Design Files into Work Library

Compile a testbench and design files into a work library on the Compile menu by clicking **Compile All** or clicking the **Compile All** toolbar icon on the Compile menu.

Compile Testbench and Design Files into Work Library Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vlog -work work <my_test bench.v> <my_design_files.v>↵
```



Resolve compile-time errors before proceeding to the following section.

Loading the Design

Perform the following steps to load a design:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. Click the **Libraries** tab.
3. In the **Search Libraries** box, click **Add**.
4. Specify the location of the **lpm** or **altera_mf** simulation libraries.



If you are using the ModelSim-Altera version, refer to Table 2–3 on page 2–5 and Table 2–4 on page 2–6 for the location of the precompiled simulation libraries. If you are using the Mentor Graphics ModelSim software version, browse to the library that was created earlier.

5. In the **Load Design** dialog box, click the **Design** tab and expand the work library.
6. Select the top-level design unit (your testbench).
7. In the **Resolution** list, select **ps**.
8. Click **OK**.

Loading a Design Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vsim -L altera_mf -L lmp work.<my_test bench> -t ps ↵
```

Running the Simulation

Perform the following steps to run a simulation:

1. On the View menu, point to Debug Windows and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to Debug Windows and click **Wave**.
3. Drag the signals to monitor from the Objects window and drop them into the Wave window.
4. Type the following command at the ModelSim command prompt:

   `run` <*time period*> ↵

Running the Simulation Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
add wave /<signal name> ↵
run <time period> ↵
```

Verilog HDL Functional RTL Simulation with Altera Memory Blocks

Both ModelSim software products support simulating Altera memory megafunctions initialized with Hexadecimal (Intel-Format) File (**.hex**) or RAM initialization files (**.rif**).

Although synthesis is able to read a Memory Initialization File (**.mif**), this memory file is not supported with third-party tools and must be converted to either a Hexadecimal (Intel-Format) File or RAM Initialization File.

Table 2–6 summarizes the different types of memory initialization file formats that are supported with each RTL language.

Table 2–6. Simulation Support for Memory Initialization Files

File	Verilog HDL	VHDL
Hexadecimal (Intel-Format) File	Yes *(1)*	Yes
Memory Initialization File	No	No
RAM Initialization File	Yes *(2)*	No

Notes to Table 2–6:

(1) For memories and library files from the Quartus II software version 5.0 and earlier, you must use a PLI library containing the `convert_hex2ver` function.

(2) Requires the `USE_RIF` macro to be defined, described later in this section.

To simulate your design by converting your Memory Initialization File into either a Hexadecimal (Intel-Format) File or a RAM Initialization File, perform the following steps:

1. Convert a Memory Initialization File to a Hexadecimal (Intel-Format) File or RAM Initialization File in the Quartus II software.

   Converting a Memory Initialization File to a Hexadecimal (Intel-Format) File

   a. Open the Memory Initialization File. On the File menu, click **Save As**. The **Save As** dialog box appears.

   b. In the **Save as type** list, select **Hexadecimal (Intel-Format) File (*.hex)**.

   c. Click **OK**.

   Convert a Memory Initialization File to a RAM Initialization File

   a. Open the Memory Initialization File and on the File menu, click **Export**. The **Export** dialog box appears.

   b. In the **Save as type** list, select **RAM Initialization File (*.rif)**.

c. Click **OK**.

Alternatively, you can convert a Memory Initialization File to a RAM Initialization File using the **mif2rif.exe** utility located in the <*Quartus II installation*>**/bin** directory.

```
mif2rif <mif_file> <rif_file> ↵
```

2. Modify the HDL file generated by the MegaWizard® Plug-In Manager.

   The Altera memory custom megafunction variation file includes the `lpm_file` parameter for LPM memories such as `LPM_ROM`, or the `init_file` for Altera specific memories such as an `altsyncram`, to point to the initialization file.

   In a text editor, open the custom megafunction variation file and edit the `lpm_file` or `init_file` to point to the Hexadecimal (Intel-Format) File or RAM Initialization File, as shown in the following example:

   ```
   lpm_ram_dp_component.lpm_file = "<path to HEX/RIF>"
   ```

3. Compile the functional library files with compiler directives.

   If you use a Hexadecimal (Intel-Format) File, no compiler directives are required. If you use a RAM Initialization File, you must define the **USE_RIF** macro when compiling the model library files. For example, you should enter the following when compiling the `altera_mf` library when RAM Initialization File memory initialization files are used:

   ```
   vlog -work altera_mf altera_mf.v +define+USE_RIF=1
   ```

   For the Quartus II software versions 5.0 and earlier, you must define the `NO_PLI` macro instead of `USE_RIF`. The `NO_PLI` macro is forward compatible with the Quartus II software.

Post-Synthesis Simulation

A post-synthesis simulation verifies the functionality of a design after synthesis has been performed. You can create a post-synthesis netlist in the Quartus II software and use this netlist to perform a post-synthesis simulation in ModelSim. Once the post-synthesis version of the design is verified, the next step is to place-and-route the design in the target device using the Quartus II Fitter.

Generating a Post-Synthesis Simulation Netlist

The following steps describe the process of generating a post-synthesis simulation netlist in the Quartus II software:

1. Perform Analysis and Synthesis. On the Processing menu, point to Start and click **Start Analysis and Synthesis** (you can also perform this after step 2).

2. Turn on the **Generate Netlist for Functional Simulation Only** option by performing the following steps:

    a. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.

    b. In the **Category** list, select **Simulation**. The **Simulation** page appears.

    c. In the **Tool name** list:

    - If you are using the ModelSim-Altera software, select **ModelSim-Altera**.
    - If you are using the Mentor Graphics ModelSim software, select **ModelSim**.

    d. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.

    e. Click **More Settings**. The **More EDA Tools Simulation Settings** dialog box appears. In the **Existing options settings list**, click **Generate Netlist for Functional Simulation Only** and select **On** from the **Setting** list under **Option**.

    f. Click **OK**.

    g. In the **Settings** dialog box, click **OK**.

3. Run the EDA Netlist Writer. On the Processing menu, point to Start and click **Start EDA Netlist Writer**.

   During the EDA Netlist Writer stage, the Quartus II software produces a Verilog Output File (**.vo**) or VHDL Output File (**.vhd**) that can be used for post-synthesis simulations in the ModelSim software. This netlist file is mapped to architecture-specific primitives. No timing information is included at this stage. The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to the *<project directory>***/simulation/modelsim** directory.

Simulating VHDL Designs

The following instructions help you perform a post-synthesis simulation for a VHDL design in the ModelSim software.



The following steps assume you have already created a ModelSim project.

If you are using the ModelSim-Altera software, a set of precompiled libraries are created when you install the software. Creating simulation libraries and compiling simulation models steps are not required. You can proceed directly to "Compile Testbench and Design Files into Work Library" on page 2–9.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. If you are using the Mentor Graphics ModelSim software, you must create the simulation libraries and correctly link them to your design.



This process is not required with the ModelSim-Altera version because a set of pre-compiled libraries is installed with the software.

Create Simulation Libraries Using the ModelSim GUI
Perform the following steps to create simulation libraries:

1. On the File menu, click **New Library**. The **Create a New Library** dialog box appears.

2. Select **a new Library and a logical linking to it**.

3. In the **Library Name** box, type the name of the newly created library.

4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt
Type the following commands to create simulation libraries:

```
vlib <device family name> ↵
vmap <device family name> <device family name> ↵
```

For more information about library names, refer to Table 2–9 on page 2–28.

Compile Simulation Models into Simulation Libraries Using the ModelSim GUI
Perform the following steps to compile simulation models into simulation libraries:

1. On the File menu, point to Add to Project and click **Existing File**.
2. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** directory and add the necessary gate-level simulation files to your project.
3. Select the simulation model file and on the View menu, click **Properties**.
4. In the **Compile to Library** list, select the correct library.
5. Click **OK**.
6. On the Compile menu, click **Compile selected**.

Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
vcom -work <device family name> <Quartus II installation directory> \
/eda/sim_lib/<device family name>_atoms.vhd ↵
```

```
vcom -work <device family name> <Quartus II installation directory> \
/eda/sim_lib/<device family name>_components.vhd ↵
```

Compile Testbench and VHDL Output File into Work Library

To compile the testbench and VHDL Output Files into a work library, on the Compile menu, click **Compile All** or click the **Compile All** toolbar icon on the Compile menu.

Compile Testbench and VHDL Output File into Work Library Using ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vcom -work work <my_test bench.vhd> <my_vhdl_output_file.vho>↵
```



Resolve any compilation errors before proceeding to the following section.

Loading the Design

Perform the following steps to load a design:

1. On the Simulate menu, click **Simulate**.
2. Click the **Design** tab.
3. In the **Library** list, select the **work** library.
4. In the **Simulate** dialog box, expand the **work** library and select the top-level design unit (your testbench).
5. Click **OK**.

Loading the Design Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vsim work.<my test bench> -t 1ps ↵
```



Set the time scale resolution to 1 ps when simulating Altera FPGA designs.

Running the Simulation

Perform the following steps to run a simulation:

1. On the View menu, point to Debug Windows and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to Debug Windows and click **Wave**.
3. Drag the signals to monitor from the Objects window and drop them into the Wave window.
4. Type the following command at the ModelSim command prompt:

   ```
   run <time period> ↵
   ```

Running the Simulation Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
add wave /<signal name> ↵
run <time period> ↵
```

Simulating Verilog HDL Designs

The following sections provide step-by-step instructions for performing post-synthesis simulation for Verilog HDL designs in the ModelSim software.

Create Simulation Libraries

The following steps assume you have already created a ModelSim project.



If you are using the ModelSim-Altera software, a set of precompiled libraries are created when you install the software. Creating simulation libraries and compiling simulation models steps are not required. You can proceed directly to "Compile Testbench and Design Files into Work Library" on page 2–9.

Create Simulation Libraries Using the ModelSim GUI
Perform the following steps to create simulation libraries:

1. In the ModelSim software, on the File menu, point to New and click **Library**. The **Create a New Library** dialog box appears.
2. Select **a new library and a logical mapping to it**.



The name of the libraries should be **altera_mf** (for Altera megafunctions) and **lpm** (for lpm and MegaWizard Plug-in Manager-generated entities).

3. In the **Library Name** box, type the name of the newly created library.
4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
vlib <device family name> ↵
vmap <device family name> <device family name> ↵
```

For more information about library names, refer to Table 2–9 on page 2–28.

Compile Simulation Models into Simulation Libraries Using the ModelSim GUI
Perform the following steps to compile simulation models into simulation libraries:

1. On the File menu, click **Add to Project**, then select **Existing File**.
2. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** directory and add the necessary simulation model files to your project.
3. Select the simulation model file and on the View menu, click **Properties**.
4. Specify the correct library in the **Compile to Library** box.

Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vlog -work <device family name> <Quartus II installation \
directory> /eda/sim_lib/<device family name>_atoms.v ↵
```

Compile Testbench and Verilog Output File into Work Library

To compile the testbench and Verilog Output Files into a work library, on the Compile menu, click **Compile All** or click the **Compile All** toolbar icon on the Compile menu.

Compile Testbench and Verilog Output File into Work Library Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vlog -work work <my_test bench.v> <my_verilog_output_file.vo> ↵
```



Resolve any compilation errors before proceeding to the following section.

Loading the Design

Perform the following steps to load a design:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. Click the **Libraries** tab.
3. In the **Search Libraries** box, click **Add**.
4. Specify the location of the device family simulation libraries.
5. In the **Load Design** dialog box, click the **Design** tab and expand the work library.
6. Select the top-level design unit (your testbench).
7. In the **Resolution** list, select **ps**.
8. Click **OK**.

Loading the Design Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

`vsim -L` <*gate-level simulation library*> `work.`<*my_test bench*> `-t 1ps` ↵



Set the time scale resolution to 1 ps when simulating Altera FPGA designs.

Running the Simulation

Perform the following steps to run a simulation:

1. In the View menu, point to Debug Windows and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to Debug Windows and click **Wave**.
3. Drag the signals to monitor from the Objects window and drop them into the Wave window.
4. Type the following command at the ModelSim command prompt:

   `run` <*time period*> ↵

Running the Simulation Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
add wave /<signal name> ↵
run <time period> ↵
```

Gate-Level Timing Simulation

Gate-level timing simulation is a post place-and-route simulation to verify the operation of the design after the worst-case timing delays have been calculated. This section provides detailed instructions on how to perform gate-level timing simulation in the ModelSim-Altera software and highlights differences in performing similar steps in the Mentor Graphics ModelSim software versions for VHDL and Verilog HDL designs.

Generating a Gate-Level Timing Simulation Netlist

To perform gate-level timing simulation, the ModelSim-Altera software requires information about how the design was placed into device-specific architectural blocks. The Quartus II software provides this information in the form of a Verilog Output File for Verilog HDL designs and a VHDL Output File for VHDL designs. The accompanying timing information is stored in the Standard Delay Format Output File (**.sdo**), which annotates the delay for the elements found in the Verilog Output File or VHDL Output File.

The following steps describe the process of generating a gate-level timing simulation netlist in the Quartus II software:

1. Perform a full compilation. On the Processing menu, click **Start Compilation**.
2. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
3. In the **Category** list, click the "+" icon to expand **EDA Tool Settings** and select **Simulation**. The **Simulation** page appears.
4. In the **Tool name** list:
   - If you are using the ModelSim-Altera software, select **ModelSim-Altera**.
   - If you are using the Mentor Graphics ModelSim software, select **ModelSim**.

5. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.

6. Click **OK**.

7. In the **Settings** dialog box, click **OK**.

8. Run the EDA Netlist Writer. On the Processing menu, point to Start and click **Start EDA Netlist Writer**.

   During the EDA Netlist Writer stage, the Quartus II software produces a Verilog Output File (**.vo**), VHDL Output File (**.vho**), and a SDO used for gate-level timing simulations in the ModelSim software. This netlist file is mapped to architecture-specific primitives. The timing information for the netlist is included in the SDO. The resulting netlist is located in the output directory you specified in the Settings dialog box, which defaults to the *<project directory>***/simulation/modelsim** directory.

Generating a Different Timing Model

If you enable the Quartus II Classic or Quartus II TimeQuest Timing Analyzer when generating the SDO file, slow-corner (worst case) timing models are used by default. To generate the SDO file using a different timing model, you must run the Quartus II Classic or the Quartus II TimeQuest Timing Analyzer with a different timing model before you start the EDA Netlist writer.

To run the Quartus II Classic Timing Analyzer with the best-case model, on the Processing menu, point to Start and click **Start Classic Timing Analyzer (Fast Timing Model)**. After timing analysis is complete, the Compilation Report appears. You can also type the following command at a command prompt:

```
quartus_tan <project_name> --fast_model=on ↵
```

To run the Quartus II TimeQuest Timing Analyzer with a best-case model, use the `-fast_model` option after you create the timing netlist. The following command enables the fast timing models:

```
create_timing_netlist -fast_model ↵
```

You can also type the following command at a command prompt:

```
quartus_sta <project_name> --fast_model=on ↵
```



For more information about generating the timing model, refer to the *Quartus II Classic Timing Analyzer* or *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

After you run the Classic or TimeQuest Timing Analyzer, you can perform steps 2 through 8 in "Generating a Gate-Level Timing Simulation Netlist" on page 2–23 to generate the SDO file. For fast corner timing models, the _fast post fix is added to the VO, VHO, and SDO file (for example, **my_project_fast.vo**, **my_project_fast.vho**, and **my_project_fast.sdo**).

Operating Condition Example: Generate All Timing Models for Stratix III Devices

In Stratix III and Cyclone III devices, you can specify different temperature and voltage parameters to generate the timing models. Table 2–7 shows the available operation conditions (model, voltage, and temperature) for Stratix III and Cyclone III devices.

Table 2–7. Available Operating Condition for Stratix III and Cyclone III Devices

Device Family	Model	Voltage	Temperature
Stratix III	Slow	1100 mV	85° C
	Slow	1100 mV	0° C
	Fast	1100 mV	0° C
Cyclone III	Slow	1200 mV	85° C
	Slow	1200 mV	0° C
	Fast	1200 mV	0° C

To generate the SDO files for the three different operating conditions for a Stratix III design, perform the following steps:

1. Generate all the available corner models at all operating conditions. Type the following command at a command prompt:

```
quartus_sta <project name> --multicorner ↵
```

2. Generate the ModelSim simulation output files for all three corners specified above. The output files are generated in the simulation output directory. Type the following command at a command prompt:

```
quartus_eda <project name> --simulation --tool=modelsim --format=verilog ↵
```

To summarize, for the three operating conditions the preceding steps generate the following files in the simulation output directory:

First slow corner (slow, 1100 mV, 85º C):
VO file— <*revision name*>**.vo**
SDO file— <*revision name*>**_v.sdo**

Second slow corner (slow, 1100 mV, 0º C):
VO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_slow.vo**
SDO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_v_slow.sdo**

Fast corner (fast, 1100 mV, 0º C):
VO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_fast.vo**
SDO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_v_fast.sdo**

Perform Timing Simulation Using Post-synthesis Netlist

Instead of using the gate-level netlist, you can also perform a timing simulation with the post-synthesis netlist. You can generate a SDO without running the fitter. In this case, the SDO file includes all timing values for only the device cells. Interconnect delays are not included because fitting (placement and routing) has not been performed.

To generate the post-synthesis netlist and the SDO file, type the following command at a command prompt:

```
quartus_map <project name> -c <revision name> ↵
quartus_tan <project name> -c <revision name> --post_map --zero_ic_delays ↵
quartus_eda <project name> -c <revision name> --simulation --tool= \
<3rd party EDA tool> --format=<HDL language> ↵
```

For more information on the --format and --tool options, type the following command at a command prompt:

```
quartus_eda -help=<options> ↵
```

Gate-Level Simulation Libraries

Table 2–8 provides a description of the ModelSim-Altera precompiled device libraries.

Table 2–8. ModelSim-Altera Precompiled Device Libraries

Library	Description
arriagx_hssi	Precompiled library for Arria® GX device designs using the Gigabit Transceiver Block (alt2gxb megafunction). This precompiled library is required for both functional and timing simulations.
stratixiii	Precompiled library for Stratix® III device designs.
stratixii	Precompiled library for Stratix II device designs.
stratixiigx	Precompiled library for Stratix II GX device designs.
stratixiigx_hssi	Precompiled library for Stratix II GX device designs using the Gigabit Transceiver Block (`alt2gxb` megafunction). This precompiled library is required for both functional and timing simulations.
stratix	Precompiled library for Stratix device designs.
stratixgx	Precompiled library for Stratix GX device designs.
stratixgx_gxb	Precompiled library for Stratix GX device designs using the Gigabit Transceiver Block. This precompiled library should be used for post-fit (timing) simulations.
altgxb	Precompiled library for Stratix GX device designs that include the `altgxb` megafunction. This precompiled library should be used for functional simulations.
cycloneii	Precompiled library for Cyclone® II device designs.
cyclone	Precompiled library for Cyclone device designs.
maxii	Precompiled library for MAX® II device designs.
max	Precompiled library for MAX 7000 and MAX 3000 device designs.
apexii	Precompiled library for APEX™ II device designs.
apex20k	Precompiled library for APEX 20K device designs.
apex20ke	Precompiled library for APEX 20KC, APEX 20KE, and Excalibur™ device designs.
mercury	Precompiled library for Mercury™ device designs.
flex10ke	Precompiled library for FLEX® 10KE and ACEX® 1K device designs.
flex6000	Precompiled library for FLEX 6000 device designs.

Table 2–9 shows the location of the timing simulation libraries in the ModelSim-Altera software for Verilog HDL.

Table 2–9. Location of Timing Simulation Libraries for ModelSim-Altera for Verilog HDL

Library	Verilog HDL
arriagx_ver	*<ModelSim-Altera installation directory>*\altera\verilog\arriagx\
arriagx_hssi_ver	*<ModelSim-Altera installation directory>*\altera\verilog\arriagx_hssi\ *(1)*
stratixii_ver	*<ModelSim-Altera installation directory>*\altera\verilog\stratixii\
stratixiigx_ver	*<ModelSim-Altera installation directory>*\altera\verilog\stratixiigx\
stratixiigx_hssi_ver	*<ModelSim-Altera installation directory>*\altera\verilog\stratixiigx_hssi\ *(1)*
stratixiii_ver	*<ModelSim-Altera installation directory>*\altera\verilog\stratixiii\
stratix_ver	*<ModelSim-Altera installation directory>*\altera\verilog\stratix\
stratixgx_ver	*<ModelSim-Altera installation directory>*\altera\verilog\stratixgx\
stratixgx_gxb_ ver	*<ModelSim-Altera installation directory>*\altera\verilog\stratixgx_gxb\
cycloneiii_ver	*<ModelSim-Altera installation directory>*\altera\verilog\cycloneiii\
cycloneii_ver	*<ModelSim-Altera installation directory>*\altera\verilog\cycloneii\
cyclone_ver	*<ModelSim-Altera installation directory>*\altera\verilog\cyclone\
maxii_ver	*<ModelSim-Altera installation directory>*\altera\verilog\maxii\
max_ver	*<ModelSim-Altera installation directory>*\altera\verilog\max\
apexii_ver	*<ModelSim-Altera installation directory>*\altera\verilog\apexii\
apex20k_ver	*<ModelSim-Altera installation directory>*\altera\verilog\apex20k\
apex20ke_ver	*<ModelSim-Altera installation directory>*\altera\verilog\apex20ke\
mercury_ver	*<ModelSim-Altera installation directory>*\altera\verilog\mercury\
flex10ke_ver	*<ModelSim-Altera installation directory>*\altera\verilog\flex10ke\
flex6000_ver	*<ModelSim-Altera installation directory>*\altera\verilog\flex6000\

Note to Table 2–9:
(1) The **stratixiigx_hssi** precompiled library is required for functional and timing simulations.

Table 2–10 shows the location of the timing simulation libraries in the ModelSim-Altera software for VHDL.

Table 2–10. Location of Timing Simulation Library Files for ModelSim-Altera for VHDL (Part 1 of 2)

Library	VHDL
arriagx	*<ModelSim-Altera installation directory>*\altera\vhdl\arriagx\
arriagx_hssi	*<ModelSim-Altera installation directory>*\altera\vhdl\arriagx_hssi\ *(1)*
stratixii	*<ModelSim-Altera installation directory>*\altera\vhdl\stratixii\

Table 2–10. Location of Timing Simulation Library Files for ModelSim-Altera for VHDL (Part 2 of 2)

Library	VHDL
stratixiigx	*<ModelSim-Altera installation directory>*\altera\vhdl\stratixiigx\
stratixiigx_hssi	*<ModelSim-Altera installation directory>*\altera\vhdl\stratixiigx_hssi\ *(1)*
stratixiii	*<ModelSim-Altera installation directory>*\altera\vhdl\stratixiii\
stratix	*<ModelSim-Altera installation directory>*\altera\vhdl\stratix\
stratixgx	*<ModelSim-Altera installation directory>*\altera\vhdl\stratixgx\
stratixgx_gxb	*<ModelSim-Altera installation directory>*\altera\vhdl\stratixgx_gxb\
cycloneiii	*<ModelSim-Altera installation directory>*\altera\vhdl\cycloneiii\
cycloneii	*<ModelSim-Altera installation directory>*\altera\vhdl\cycloneii\
cyclone	*<ModelSim-Altera installation directory>*\altera\vhdl\cyclone\
maxii	*<ModelSim-Altera installation directory>*\altera\vhdl\maxii\
max	*<ModelSim-Altera installation directory>*\altera\vhdl\max\
apexii	*<ModelSim-Altera installation directory>*\altera\vhdl\apexii\
apex20ke	*<ModelSim-Altera installation directory>*\altera\vhdl\apex20ke\
apex20k	*<ModelSim-Altera installation directory>*\altera\vhdl\apex20k\
flex10ke	*<ModelSim-Altera installation directory>*\altera\vhdl\flex10ke\
flex6000	*<ModelSim-Altera installation directory>*\altera\vhdl\flex6000\
mercury	*<ModelSim-Altera installation directory>*\altera\vhdl\mercury\

Note to Table 2–10:
(1) The **stratixiigx_hssi** precompiled library is required for functional and timing simulations.

If you are using the Mentor Graphics ModelSim software version for your timing simulation, libraries are available in the Quartus II software in the *<Quartus II installation directory>***eda****sim_lib**\ directory. Mentor Graphics ModelSim software users must use the files provided with the Quartus II software.

Simulating VHDL Designs

The following section provides step-by-step instructions for performing gate-level timing simulation for VHDL designs. The following steps assume you have already created a ModelSim project. For additional information, refer to "Altera Design Flow with ModelSim or ModelSim-Altera Software" on page 2–3.



If you are using the ModelSim-Altera software, a set of precompiled libraries are created when you install the software. Creating simulation libraries and compiling simulation models steps are not required. You can proceed directly to "Compile Testbench and Design Files into Work Library" on page 2–9.

Create Simulation Libraries

If you are using the Mentor Graphics ModelSim software, create the gate-level simulation libraries and correctly link them to your design.

Create Simulation Libraries Using the ModelSim GUI
Perform the following steps to create simulation libraries:

1. In the ModelSim software, on the File menu, point to New and click **Library**. The **Create a New Library** dialog box appears.
2. Select **a new library and a logical mapping to it**.



The name of the libraries should be **altera_mf** (for Altera megafunctions) and **lpm** (for lpm and MegaWizard Plug-In Manager-generated entities).

3. In the **Library Name** box, type the name of the newly created library.



The library name must be one of those listed in Table 2–10 on page 2–28.

4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
vlib <device family name> ↵
vmap <device family name> <device family name> ↵
```

For more information about library names, refer to Table 2–9 on page 2–28.

Compile Simulation Models into Simulation Libraries Using the ModelSim GUI
Perform the following steps to compile simulation models into simulation libraries:

1. On the File menu, point to Add to Project and click **Existing File**.
2. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** directory, and add the necessary gate-level simulation files to your project.
3. Select the simulation model file, and on the View menu, click **Properties**.
4. In the **Compile to Library** list, select the correct library.
5. Click **OK**.
6. On the Compile menu, click **Compile selected**.

Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
vcom -work <device family name> <Quartus II installation directory> \
/eda/sim_lib/<device family name>_atoms.vhd ↵

vcom -work <device family name> <Quartus II installation directory> \
/eda/sim_lib/<device family name>_components.vhd ↵
```

Compile Testbench and VHDL Output File into Work Library

Compile testbench and VHDL Output Files into a work library on the Compile menu by clicking **Compile All** or by clicking the **Compile All** toolbar icon on the Compile menu.

Compile Testbench and VHDL Output File into Work Library Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vcom -work work <my_test bench.vhd> <my_vhdl_output_file.vho> ↵
```



Resolve any compilation errors before proceeding to the following section.

Loading the Design

Perform the following steps to load a design:

1. On the Simulate menu, click **Start Simulation**.
2. Click the **SDF** tab, and click **Add**.
3. In the **Add SDF Entry** dialog box, click **Browse** and select the Standard Delay Format Output File (**.sdo**).
4. In the **Apply to Region** dialog box, type in the instance path to which the SDO should be applied. For example, if you are using a testbench exported into the Quartus II software from a Vector Waveform File, the instance path should be set to `/i1`.

   You do not have to choose from the **Delay** list because the Quartus II EDA Netlist Writer generates the SDO using the same value for the triplet (minimum, typical, and maximum timing values). The value is derived from either the fast (minimum) timing model or worst case (maximum) timing model, depending on which timing model was used in the last timing analysis. In the standard compilation flow, the Quartus II software writes the SDO using timing values from the worst case (maximum) timing model.
5. Click **OK**.
6. Click the **Design** tab. In the **Resolution** list, select **ps**.
7. In the **Library** list, select the **work** library.
8. In the **Start Simulation** dialog box, expand the **work** library.
9. Select the top-level design unit (your testbench).
10. Click **OK**.

Loading a Design Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vsim -sdftyp <instance path to design>=<path to SDO> work. \
<my_test bench> -t ps ↵
```

Running the Simulation

Perform the following steps to run a simulation:

1. In the View menu, point to Debug Windows and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to Debug Windows and click **Wave**.
3. Drag the signals to monitor from the Objects window and drop them into the Wave window.
4. Type the following command at the ModelSim command prompt:

   `run <time period> ↵`

Running a Simulation Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
add wave /<signal name> ↵
run <time period> ↵
```

Simulating Verilog HDL Designs

The following sections provide step-by-step instructions on performing gate-level timing simulation for Verilog HDL designs in the ModelSim-Altera software.



If you are using the ModelSim-Altera software, a set of pre-compiled libraries are created when you install the software. Creating simulation libraries and compiling simulation models steps are not required. You can proceed directly to "Compile Testbench and Design Files into Work Library" on page 2–9.

Create Simulation Libraries

If you are using the Mentor Graphics ModelSim software, you must create the simulation libraries and correctly link them to your design.

The following steps assume you have already created a ModelSim project. For additional information, refer to "Altera Design Flow with ModelSim or ModelSim-Altera Software" on page 2–3.

Create Simulation Libraries Using the ModelSim GUI
Perform the following steps to create simulation libraries:

1. In the ModelSim software, on the File menu, point to New and click **Library**. The **Create a New Library** dialog box appears.
2. Select **a new library and a logical mapping to it**.

   The names of the libraries should be **altera_mf** (for Altera megafunctions) and **lpm** (for lpm and MegaWizard Plug-In Manager-generated entities).

3. In the **Library Name** box, type the name of the newly created library.
4. Click **OK**.

Create Simulation Libraries Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

```
vlib <library name> ↵
vmap <library name> <device family name> ↵
```

For more information about library names, refer to Table 2–9 on page 2–28.

Compile Simulation Models into Simulation Libraries Using the ModelSim GUI
Perform the following steps to compile simulation models into simulation libraries:

1. On the File menu, point to Add to Project and click **Existing File**.
2. Browse to the <*Quartus II installation directory*>**/eda/sim_lib**, and add the necessary simulation model files to your project.
3. Select the simulation model file, and on the View menu, click **Properties**.
4. In the **Compile to Library** list, select the correct library.

5. Click **OK**.

6. On the Compile menu, click **Compile selected**.

Compile Simulation Models into Simulation Libraries Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vlog -work <device family name> <Quartus II installation directory> /eda/sim_lib/<device family name> \
_atoms.v ↵
```

Compile Testbench and Verilog Output File into Work Library

Compile a testbench and Verilog Output File into a work library on the Compile menu by clicking **Compile All** or by clicking the **Compile All** toolbar icon on the Compile menu.

Compile Testbench and Verilog Output File into Work Libraries Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

```
vlog -work work <my_test bench.v> <my_verilog_output_file.vo> ↵
```



Resolve any compilation errors before proceeding to the following section.

Loading the Design

Perform the following steps to load a design:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.

2. Click the **Libraries** tab.

3. In the **Search Libraries** box, click **Add**.

4. Specify the location of the **lpm** or **altera_mf** simulation libraries.



If you are using the ModelSim-Altera version, refer to Table 2–3 on page 2–5 and Table 2–4 on page 2–6 for the location of the precompiled simulation libraries. If you are using the Mentor Graphics ModelSim software version, browse to the library that you created earlier.

5. In the **Load Design** dialog box, click the **Design** tab and expand the work library.

6. Select the top-level design unit (your testbench).

7. In the **Resolution** list, select **ps**.

8. Click **OK**.

When simulating in Verilog HDL, the SDO does not have to be manually specified because in the Quartus II generated Verilog Output File, there is a `$sdf_annotate` task that ModelSim uses to look into the current directory from which VSIM was run and uses to look for the SDO. If your SDO is not in the same directory from which you ran VSIM, you can either copy the SDO into your current directory or comment out the `$sdf_annotate` line in the Verilog Output File and manually specify the SDO in the Load Design dialog box.

Loading the Design Using the ModelSim Command Prompt
Type the following command at the ModelSim command prompt:

`vsim -L` <*location of the gate level simulation library*> `-work.`<*my_test bench*> `-t ps` ↵

Running the Simulation

Perform the following steps to run a simulation:

1. On the View menu, point to Debug Windows and click **Objects**. This command displays all objects in the current scope.

2. On the View menu, point to Debug Windows and click **Wave**.

3. Drag the signals to monitor from the Objects window and drop them into the Wave window.

4. Type the following command at the ModelSim command prompt:

   `run` <*time period*> ↵

Running the Simulation Using the ModelSim Command Prompt
Type the following commands at the ModelSim command prompt:

`add wave /`<*signal name*> ↵
`run` <*time period*> ↵

For the design examples to run gate-level timing simulation in VHDL or Verilog language, refer to:
www.altera.com/support/examples/modelsim/exm-modelsim.html.

Simulating Designs that Include Transceivers

If your design includes a Stratix GX or Stratix II GX transceiver, you must compile additional library files to perform functional or timing simulations.

Stratix GX Functional Simulation

To perform a functional simulation of your design that instantiates the `altgxb` megafunction which enables the gigabit transceiver block on Stratix GX devices, compile the **stratixgx_mf** model file into the **altgxb** library.

The **stratixiigx_mf** model file references the **lpm** and **sgate** libraries. If you are using ModelSim PE/SE, you must create these libraries to perform a simulation.

Example: Performing Functional Simulation for Stratix GX in Verilog HDL

If you are using ModelSim-Altera, compiling the libraries is not necessary. You can simulate the design directly by typing the following command:

```
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L altgxb work.<my design> ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulation the designs. Type the following commands at the ModelSim command prompt to compile and simulate the design:

```
vlib work ↵
vlib lpm ↵
vlib altera_mf ↵
vlib sgate ↵
vlib altgxb ↵
vlog -work lpm 220model.v ↵
vlog -work altera_mf altera_mf.v ↵
vlog -work sgate sgate.v ↵
vlog -work altgxb stratixgx_mf.v ↵
vsim -L lpm -L sgate-L altgxb work.<my design> ↵
```

Example: Performing Functional Simulation for Stratix GX in VHDL

If you are using ModelSim-Altera, compiling the libraries is not necessary and you can simulate the design directly by typing the following command:

```
vsim -L lpm -L sgate -L altgxb work.<my design> r ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulation the designs. Type the following commands at the ModelSim command prompt to compile and simulate the design:

```
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
vcom -work lpm 220pack.vhd 220model.vhd ↵
vcom -work sgate sgate_pack.vhd sgate.vhd ↵
vcom -work altgxb stratixgx_mf.vhd stratixgx_mf_components.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L altgxb work.<my design> ↵
```

Stratix GX Post-Fit (Timing) Simulation

Perform a post-fit timing simulation of your design that includes a Stratix GX transceiver by compiling the **stratixgx_atoms** and **stratixgx_hssi_atoms** model files into the **stratixgx** and **stratixgx_gxb** libraries, respectively.



The **stratixgx_hssi_atoms** model file references the **lpm** and **sgate** libraries. If you are using ModelSim PE/SE, you must create these libraries to perform a simulation.

Example: Performing Timing Simulation for Stratix GX in Verilog HDL

If you are using ModelSim-Altera, compiling the libraries is not necessary. You can simulate the design directly by typing the following command:

```
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_ver -L \
stratixgx_gxb work.<my design> -t ps +transport_int_delays \
+transport_path_delays ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. Type the following commands at the ModelSim command prompt to compile and simulate the design:

```
vlog -work lpm 220model.v ↵
vlog -work altera_mf altera_mf.v ↵
vlog -work sgate sgate.v ↵
vlog -work stratixgx stratixgx_atoms.v ↵
vlog -work stratixgx_gxb stratixgx_hssi_atoms.v ↵
vsim -L lpm -L altera_mf -L sgate -L stratixgx -L stratixgx_gxb \
work.<my design> -t ps +transport_int_delays +transport_path_delays ↵
```



This example assumes you are using ModelSim PE/SE. If you are using ModelSim-Altera, type the following command to simulate your design:

```
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_ver -L \
stratixgx_gxb work.<my design> -t ps +transport_int_delays \
+transport_path_delays ↵
```

Example: Performing Timing Simulation for Stratix GX in VHDL

If you are using ModelSim-Altera, compiling the libraries is not necessary. You can simulate the design directly by typing the following command:

```
vsim -L lpm -L altera_mf -L sgate -L stratixgx -L stratixgx_gxb \
work. <my design> -t ps - +transport_int_delays+transport_path_delays ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. Type the following commands at the ModelSim command prompt to compile and simulate the design:

```
vcom -work lpm 220pack.vhd 220model.vhd ↵
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
vcom -work sgate sgate_pack.vhd sgate.vhd ↵
vcom -work stratixgx stratixgx_atoms.vhd stratixgx_components.vhd ↵
vcom -work stratixgx_gxb stratixgx_hssi_atoms.vhd \
stratixgx_hssi_components.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixgx -L stratixgx_gxb \
work. <my design> -t ps +transport_int_delays +transport_path_delays ↵
```



This example assumes you are using ModelSim PE/SE. If you are using ModelSim-Altera, type the following command to simulate your design:

```
vsim -L lpm -L altera_mf -L sgate -L stratixgx -L stratixgx_gxb \
work. <my design> -t ps - +transport_int_delays+transport_path_delays ↵
```

Stratix II GX Functional Simulation

To perform a functional simulation of your design that instantiates the `alt2gxb` megafunction, which enables the gigabit transceiver block on Stratix II GX devices, compile the **stratixiigx_hssi** model file into the **stratixiigx_hssi** library.

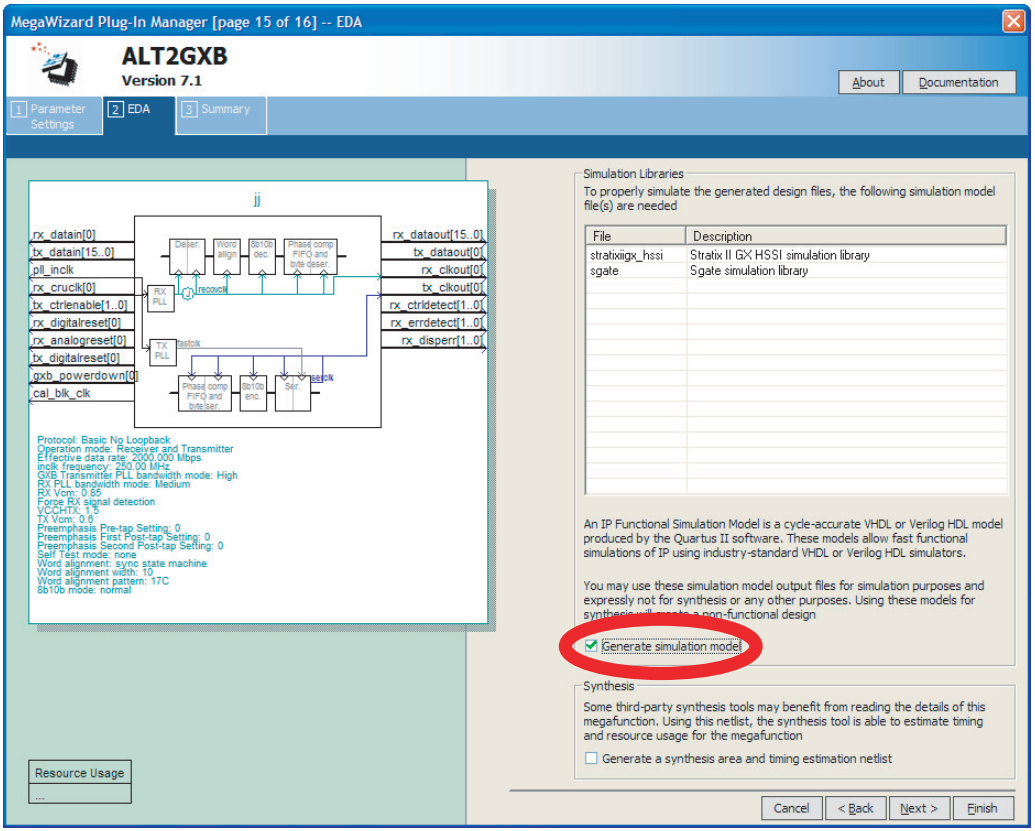


The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries. If you are using ModelSim PE/SE, you must create these libraries to perform a simulation.

Generate a functional simulation netlist by turning on **Generate Simulation Model** in the Simulation Library tab of the `alt2gxb` MegaWizard Plug-In Manager (Figure 2–2). The *<alt2gxb entity name>*.**vho** or *<alt2gxb module name>*.**vo** is generated in the current project directory.

The Quartus II-generated `alt2gxb` functional simulation library file references `stratixiigx_hssi` wysiwyg atoms.

Figure 2–2. alt2gxb MegaWizard



Example: Performing Functional Simulation for Stratix II GX in Verilog HDL

If you are using ModelSim-Altera, compiling the libraries is not necessary and you can simulate the design directly by typing the following command:

```
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_hssi_ver \
work.<my design> ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. Type the following commands at the ModelSim command prompt to compile and simulate the design:

```
vlog -work lpm 220model.v ↵
vlog -work altera_mf altera_mf.v ↵
vlog -work sgate sgate.v ↵
vlog -work stratixiigx_hssi stratixiigx_hssi_atoms.v ↵
vlog -work work <alt2gxb module name>.vo ↵
vsim -L lpm -L altera_mf -L sgate -L stratixgx_hssi work.<my design> ↵
```

Example: Performing Functional Simulation for Stratix II GX in VHDL

If you are using ModelSim-Altera, compiling the libraries is not necessary. You can simulate the design directly by typing the following command:

```
vsim -L lpm -L altera_mf -L sgate -L stratixgx_hssi work.<my design> r ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. Type the following commands at the ModelSim command prompt to compile and simulate the design:

```
vcom -work lpm 220pack.vhd 220model.vhd ↵
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
vcom -work sgate sgate_pack.vhd sgate.vhd ↵
vcom -work stratixiigx_hssi stratixiigx_hssi_components.vhd \
stratixiigx_hssi_atoms.vhd ↵
vcom -work work <alt2gxb entity name>.vho ↵
vsim -L lpm -L altera_mf -L sgate -L stratixgx_hssi work.<my design> ↵
```

Stratix II GX Post-Fit (Timing) Simulation

To perform a post-fit timing simulation of your design that includes a Stratix II GX transceiver, compile `stratixiigx_atoms` and `stratixiigx_hssi_atoms` into the **stratixiigx** and **stratixiigx_hssi** libraries, respectively.

The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries. If you are using ModelSim PE/SE, you must create these libraries to perform a simulation.

Example: Performing Timing Simulation for Stratix II GX in Verilog HDL

If you are using ModelSim-Altera, compiling the libraries is not necessary and you can simulate the design directly by typing the following command:

```
vsim -L lpm -L altera_mf -L sgate -L stratixiigx -L stratixiigx_hssi \
work.<my design> -t ps +transport_int_delays +transport_path_delays ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. Type the following commands at the ModelSim command prompt to compile and simulate the design:

```
vlog -work lpm 220model.v ↵
vlog -work altera_mf altera_mf.v ↵
vlog -work sgate sgate.v ↵
vlog -work stratixiigx  stratixiigx_atoms.v ↵
vlog -work stratixiigx_hssi stratixiigx_hssi_atoms.v ↵
vsim -L lpm -L altera_mf -L sgate -L stratixiigx -L stratixiigx_hssi \
work.<my design> -t ps +transport_int_delays +transport_path_delays ↵
```

Example: Performing Timing Simulation for Stratix II GX in VHDL

If you are using ModelSim-Altera, compiling the libraries is not necessary. You can simulate the design directly by typing the following command:

```
vsim -L lpm -L altera_mf -L sgate -L stratixiigx -L stratixiigx_hssi \
work.<mydesign> -t ps +transport_int_delays +transport_path_delays ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. Type the following commands at the ModelSim command prompt to compile and simulate the design:

```
vcom -work lpm 220pack.vhd 220model.vhd ↵
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
vcom -work sgate sgate_pack.vhd sgate.vhd ↵
vcom -work stratixiigx  stratixiigx_atoms.vhd stratixiigx_components.vhd ↵
vcom -work stratixiigx_hssi stratixiigx_hssi_components.vhd \
stratixiigx_hssi_atoms.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixiigx -L stratixiigx_hssi \
work.<my design> -t ps +transport_int_delays +transport_path_delays ↵
```

This example assumes you are using ModelSim PE/SE. If you are using ModelSim-Altera, you do not need to compile any libraries and can type the following command:

```
vsim -L lpm -L altera_mf -L sgate -L stratixiigx -L stratixiigx_hssi \
work.<my design> -t ps +transport_int_delays +transport_path_delays ↵
```

Transport Delays

By default, the ModelSim software filters out all pulses that are shorter than the propagation delay between primitives. Turning on the transport delay options in the ModelSim software prevents the simulation tool from filtering out these pulses. Use the following options to ensure that all signal pulses are seen in the simulation results.

+transport_path_delays

Use this option when the pulses in your simulation may be shorter than the delay within a gate-level primitive.

+transport_int_delays

Use this option when the pulses in your simulation may be shorter than the interconnect delay between gate-level primitives.

The `+transport_path_delays` and `+transport_int_delays` options are also used by default in the NativeLink feature for gate-level timing simulation.

For more information about either of these options, refer to the ModelSim Altera Command Reference installed with the ModelSim software.

The following ModelSim software command describes the command-line syntax to perform a gate-level timing simulation with the device family library:

```
vsim -t 1ps -L stratixii -sdftyp /i1=filtref_vhd.sdo work.filtref_vhd_vec_tst \
+transport_int_delays +transport_path_delays
```

Using the NativeLink Feature with ModelSim

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools and allows you to run ModelSim within the Quartus II software.

Setting Up NativeLink

To run ModelSim automatically from the Quartus II software using the NativeLink feature, you must specify the path to your simulation tool by performing the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, select **EDA Tool Options**.
3. Double-click the entry under **Location of executable** beside the name of your EDA Tool.
4. Type or browse to the directory containing the executables of your EDA tool.



For ModelSim-Altera and ModelSim SE/PE, executable files are stored in the **win32aloem** and **win32** directories, respectively.

c:\<*ModelSim-Altera installation path***>\win32aloem**

c:\<*ModelSim installation path***>\win32**

5. Click **OK**.

You can also specify the path to the simulator's executables by using the `set_user_option` TCL command:

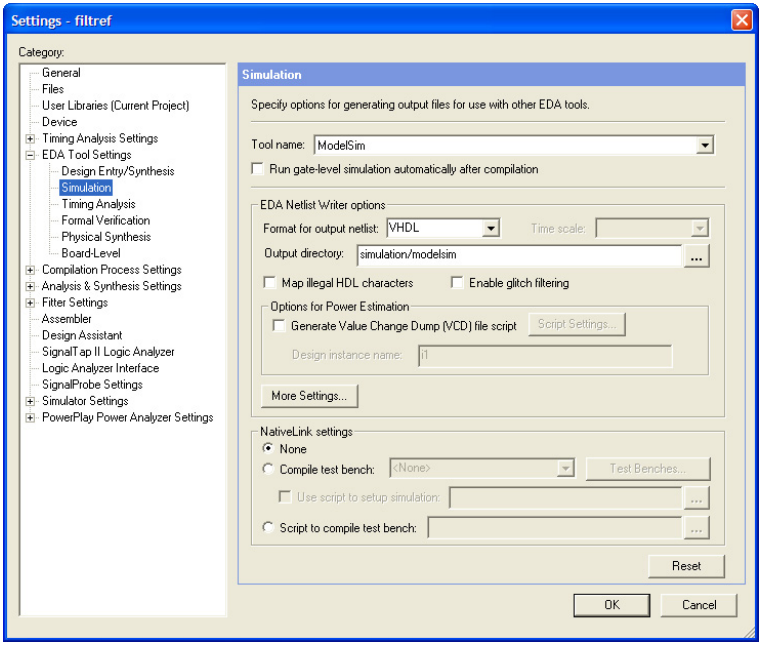
```
set_user_option -name EDA_TOOL_PATH_MODELSIM <path to executables>
set_user_option -name EDA_TOOL_PATH_MODELSIM_ALTERA <path to executables>
```

Performing an RTL Simulation Using NativeLink

To run a functional RTL simulation with the ModelSim software in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears (Figure 2–3).

Figure 2–3. Simulation Page in the Settings Dialog Box



3. In the **Tool name** list, select one of the following choices:
   - ModelSim
   - ModelSim-Altera
4. If your design is written entirely in Verilog HDL or in VHDL, the NativeLink feature automatically chooses the correct language and Altera simulation libraries. If your design is written with mixed languages, the NativeLink feature uses the default language specified in the **Format for output netlist** list. To change the default language when there is a mixed language design, under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. Table 2–11 shows the design languages for output netlists and simulation models.

Table 2–11. NativeLink Design Languages

Design File	Format for Output Netlist	Simulation Models Used
Verilog	Any	Verilog
VHDL	Any	VHDL
Mixed	Verilog	Verilog
Mixed	VHDL	VHDL



For mixed language simulation, choose the same language that was used to generate your megafunctions to ensure correct parameter passing between the megafunctions and the Altera libraries. For example, if your `altsyncram` megafunction was generated in VHDL, choose VHDL as the format for the output netlist.

When creating mixed language designs, it is important to be aware of the following:

- EDA Simulation tools do not allow seamless passing of parameters when a VHDL entity is instantiated in Verilog designs.
- The ModelSim and ModelSim-Altera software do not allow the use of Verilog User Defined Primitives (UDPs) to be instantiated in VHDL designs.

5. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.

   For more information about setting up a testbench with NativeLink, refer to "Setting Up a Testbench" on page 2–48.

6. Click **OK.**

7. On the Processing menu, point to Start and click **Start Analysis and Elaboration** to perform an analysis and elaboration. This command collects all your file name information and builds your design hierarchy in preparation for simulation.

8. On the Tools menu, point to **EDA Simulation Tool** and click **Run EDA RTL Simulation** to automatically run ModelSim, compile all necessary design files, and complete a simulation.

Performing a Gate-Level Simulation Using NativeLink

To run a gate-level timing simulation with the ModelSim software in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears (Figure 2–3 on page 2–45).
3. In the **Tool name** list, select one of the following:
   - ModelSim
   - ModelSim-Altera
4. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, choose **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
5. To run a gate-level simulation after each full compilation, turn on **Run Gate Level Simulation automatically after compilation**.
6. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.
7. Click **OK**.
8. On the Processing menu, point to Start and click **Start EDA Netlist Writer** to generate a simulation netlist of your design.
9. On the Tools menu, point to EDA Simulation Tool and click **Run EDA Gate Level Simulation** to automatically run ModelSim, compile all necessary design files, and complete a simulation.



A ModelSim Macro File (***.do**) is generated in the *<project_directory>***simulation****modelsim** directory while running NativeLink. You can perform a simulation with the DO file directly from ModelSim when you rerun a simulation without using NativeLink. To perform the simulation directly without NativeLink, type the following command in the ModelSim console: **do** *<generated_do_file>*.**do**.

Setting Up a Testbench

You can use NativeLink to compile your design files and testbench files, and run an EDA simulation tool to automatically perform a simulation.

To set up NativeLink for simulation, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the "+" icon to expand **EDA Tool Settings** and select **Simulation**. The **Simulation** page appears.
3. Under **NativeLink settings**, select **None**, **Compile test bench**, or **Script to compile test bench** (Table 2–12).

Table 2–12. NativeLink Settings

Settings	Description
None	Compile simulation models and design files.
Compile test bench	NativeLink compiles simulation models, design files, testbench files, and starts simulation.
Script to compile test bench	NativeLink compiles the simulation models and design files. The script you provide is sourced after design files compile. Use this option when you want to create your own script to compile your testbench and perform simulation.

4. If you select **Compile test bench**, select your test bench setup from the **Compile test bench** list. You can use different testbench setups to specify different test scenarios. If there are no testbench setups entered, create a testbench setup by performing the following steps:
   a. Click **Test Benches**. The **Test Benches** dialog box appears.
   b. Click **New**. The **New Test Bench Settings** dialog box appears.
   c. In the **Test Bench name** box, type in the testbench setup name that identifies the different test bench setups.
   d. In the **Test bench entity** box, type in the top-level testbench entity name. For example, for a Quartus II generated VHDL testbench, type in <*Vector Waveform File name*>`_vhd_vec_tst`.
   e. In the **Instance** box, type in the full instance path to the top level of your FPGA design. For example, for a Quartus II generated VHDL testbench, type in `i1`.

f. Under **Simulation period**, select **Run simulation until all vector stimuli are used** or specify the end time of the simulation.

g. Under **Test bench files**, browse and add all your testbench files in the **File name** box. Use the **Up** and **Down** button to reorder your files. The script used by NativeLink compiles the files in order from top to bottom.



You can also specify the library name and the HDL version to compile the testbench file. Native link compiles the testbench to a library name using the specified HDL version.

h. Click **OK**.

i. In the **Test Benches** dialog box, click **OK**.

5. Under **NativeLink settings**, you can turn on **Use script to setup simulation** and browse to your script. Your script is executed to set up and run simulation after loading the design using the `vsim` command.

6. If you choose **Script to compile test bench**, browse to your script and click **OK**.

Creating a Testbench

In the Quartus II software, you can create a Verilog HDL or VHDL testbench from a Vector Waveform File. The generated testbench includes the behavior of the input stimulus and applies it to your instantiated top-level FPGA design.

1. On the File menu, click **Open**. The **Open** dialog box appears.

2. Click the **Files of type** arrow and select **Waveform/Vector Files**. Select your Vector Waveform File.

3. Click **Open**.

4. On the File menu, click **Export**. The **Export** dialog box appears.

5. Click the **Save as type** arrow and select **VHDL Test Bench File (*.vht)** or **Verilog Test Bench File (*.vt)**.

6. You can turn on **Add self-checking code to file** to check your simulation results against your Vector Waveform File.

7. Click **Export**. Your VHDL or Verilog testbench file is generated in your project directory.

Scripting Support

You can run procedures and create settings described in this chapter in a Tcl script. You can also run some procedures at the command line prompt.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For more information about command line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For detailed information about scripting command options, refer to the **Qhelp** command line and Tcl API help browser.

Type this command to start the Qhelp help browser:

```
quartus_sh --qhelp↵
```

Generating a Post-Synthesis Simulation Netlist for ModelSim

You can use the Quartus II software to generate a post-synthesis simulation netlist with Tcl commands or with a command at the command-line prompt. The following example assumes that you are selecting ModelSim (Verilog HDL output from Quartus II software).

Tcl Commands

Use the following Tcl commands to set the output format to Verilog HDL, the simulation tool to ModelSim for Verilog HDL, and to generate a functional netlist:

```
set_global_assignment-name EDA_SIMULATION_TOOL "ModelSim (Verilog)"
set_global_assignment-name EDA_GENERATE_FUNCTIONAL_NETLIST ON
```

Command Prompt

Use the following command to generate a simulation output file for the ModelSim simulator. Specify VHDL or Verilog HDL for the format:

```
quartus_eda <project name> --simulation=on --format=<format> --tool=ModelSim \
--functional ↵
```

Generating a Gate-Level Timing Simulation Netlist for ModelSim

You can use the Quartus II software to generate a gate-level timing simulation netlist with Tcl commands or with a command at the command prompt.

Tcl Commands

Use one of the following Tcl commands:

```
set_global_assignment -name EDA_SIMULATION_TOOL "ModelSim-Altera (Verilog)"
set_global_assignment -name EDA_SIMULATION_TOOL "ModelSim-Altera (VHDL)"
set_global_assignment -name EDA_SIMULATION_TOOL "ModelSim (Verilog)"
set_global_assignment -name EDA_SIMULATION_TOOL "ModelSim (VHDL)"
```

Command Line

Generate a simulation output file for the ModelSim simulator by specifying VHDL or Verilog HDL for the format by typing the following command at the command prompt:

```
quartus_eda <project name> --simulation=on --format=<format> --tool=ModelSim ↵
```

Software Licensing and Licensing Setup

License the ModelSim-Altera software with a parallel port software guard (T-guard), USB guard, FIXEDPC license, or a network FLOATNET or FLOATPC license. Each Altera software subscription includes a license for either VHDL or Verilog HDL. Network licenses with multiple users may have their licenses split between VHDL and Verilog HDL in any ratio.



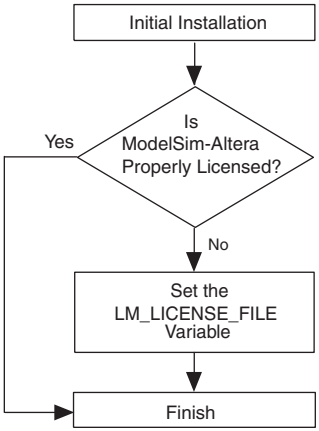
The USB software guard is not supported by versions earlier than Mentor Graphics ModelSim software 5.8d.

Obtain a license for the ModelSim-Altera software from the Altera website at **www.altera.com**. Get licensing information for the Mentor Graphics ModelSim software directly from Mentor Graphics. Refer to Figure 2–4 for the set-up process.



For ModelSim-Altera versions prior to 5.5b, use the **PCLS** utility included with the software to set up the license.

Figure 2–4. ModelSim-Altera Licensing Set Up Process



LM_LICENSE_FILE Variable

Altera recommends setting the `LM_LICENSE_FILE` environment variable to the location of the license file.

Conclusion

Using the ModelSim-Altera simulation software within the Altera FPGA design flow enables Altera software users to easily and accurately perform functional RTL simulations, post-synthesis simulations, and gate-level simulations on their designs. Proper verification of designs at the functional, post-synthesis, and post place-and-route stages using the ModelSim-Altera software helps ensure design functionality and success and, ultimately, a quick time-to-market.

Referenced Documents

This chapter references the following documents:

- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 2–13 shows the revision history for this chapter.

Table 2–13. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Updated Table 2–2. Updated "Operating Condition Example: Generate All Timing Models for Stratix III Devices" on page 2–25.	Updated for the Quartus II software version 7.2.
May 2007 v7.1.0	Updated "Functional RTL Simulation" on page 2–5. Updated "Gate-Level Timing Simulation" on page 2–23. Added "Perform Timing Simulation Using Post-synthesis Netlist" on page 2–26. Updated examples in "Simulating Designs that Include Transceivers" on page 2–37. Updated procedures in "Setting Up a Testbench" on page 2–48. Added "Referenced Documents" on page 2–52.	Updated for the Quartus II software version 7.1.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only.	—
November 2006 v6.1.0	• Added ModelSim-Altera Web Edition to Table 2-1. • Added Stratix III library support to Table 2-8, 2-9, and 2-10. • Other minor changes to chapter.	Updated for the Quartus II software version 6.1.
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0: • Added a section on setting ModelSim as the Simulation Tool • Updated EDA Tools Settings in the GUI. • Updated the Synopsys Design Constraints File information. • Updated the device information. • Added Quartus II-Generated Testbench information • Updated megafunction information.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
May 2005 v5.0.0	• Updates to tables, figures. • Updated information. • New functionality for Quartus II software 5.0.	—
December 2004 v3.0	• Reorganized chapter, updated information. • Updates to tables, figures. • New functionality for Quartus II software 4.2.	—
June 2004 v2.0	• Updates to tables, figures. • New functionality for Quartus II software 4.1.	—
February 2004 v1.0	Initial release.	—

3. Synopsys VCS Support

QII53002-7.2.0

Introduction

This chapter is an overview about using the Synopsys VCS software to simulate designs that target Altera® FPGAs. It provides a step-by-step explanation of how to perform functional register transfer level (RTL) simulations, post-synthesis simulations, and gate-level timing simulations using the VCS software.

This chapter discusses the following topics:

- "Software Requirements"
- "Common VCS Software Compiler Options" on page 3–11
- "Using VirSim" on page 3–12
- "Debugging Support Command-Line Interface" on page 3–12
- "Simulating Designs that Include Transceivers" on page 3–13
- "Using PLI Routines with the VCS Software" on page 3–16
- "Transport Delays" on page 3–17
- "Using NativeLink with the VCS Software" on page 3–18
- "Scripting Support" on page 3–23

Software Requirements

To simulate your design using VCS, you must first set up the Altera libraries. These libraries are installed with the Quartus II software.

Table 3–1 shows the compatibility between versions of the Quartus II software and the Synopsys VCS software.

Table 3–1. Supported Quartus II and VCS Software Version Compatibility

Synopsys	Altera
VCS software version Y-2006.06-SP1	Quartus II software version 7.2
VCS software version 2006.06	Quartus II software version 7.1
VCS software version 2005.06-SP2	Quartus II software version 7.0 and 6.1
VCS software version 2005.06-SP1	Quartus II software version 6.0
VCS software version 7.2	Quartus II software version 5.1
VCS software version 7.2	Quartus II software version 5.0
VCS software version 7.1.1	Quartus II software version 4.2



For more information about installing the software and the directories created during the Quartus II software installation, refer to the *Quartus II Installation and Licensing for Windows* or the *Quartus II Installation and Licensing for UNIX and Linux Workstation* manuals.

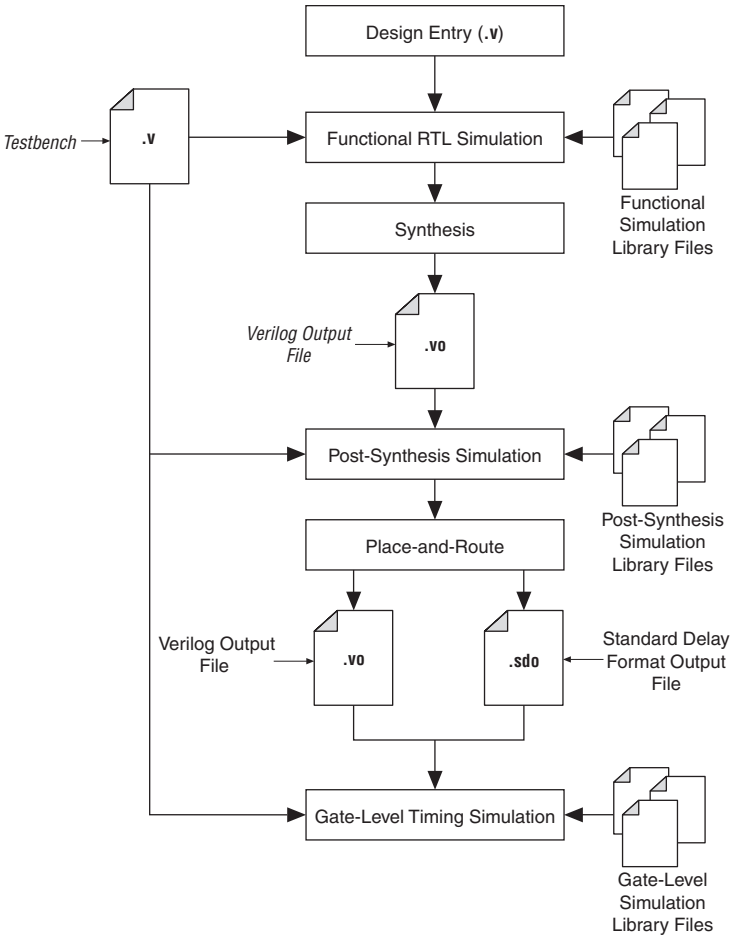
Using VCS in the Quartus II Design Flow

You can perform the following types of simulations using VCS:

- Functional RTL
- Post-synthesis
- Gate-level timing

Figure 3–1 shows the VCS and Quartus II software design flow.

Figure 3–1. Altera Design Flow with the VCS and Quartus II Software



Functional Simulations

Functional RTL simulations verify the functionality of the design before synthesis, placement, and routing. These simulations are independent of any Altera FPGA architecture implementation. Once the HDL designs are verified to be functionally correct, the next step is to synthesize the design and use the Quartus II software to place-and-route the design in an Altera device.

To functionally simulate an Altera FPGA design in the VCS software that uses Altera intellectual property (IP) megafunctions, or library of parameterized modules (LPM) functions, you must include certain libraries during the compilation. Table 3–2 summarizes the Verilog HDL library files that are required to compile LPM functions and Altera megafunctions.

Table 3–2. Altera Verilog HDL Functional/Behavioral Simulation Library Files

Library Name	Verilog HDL Libraries
LPM	**220model.v**
`altera_mf`	**altera_mf.v**
`altgxb`	**stratixgx_mf.v** *(1)*
`alt2gxb`	**stratixiigx_hssi_atoms.v** *(1)* **arriagx_hssi_atoms.v** *(1)*
`sgate`	**sgate.v**
`altera`	**altera_primitives.v**

Note to Table 3–2:
(1) The **stratixgx_mf.v**, **stratixiigx_hssi_atoms.v**, and **arriagx_hssi_atoms.v** library files require the LPM and SGATE libraries.

The library files in Table 3–2 are installed with the Quartus II software. These files are found in the *<path to Quartus II installation>*`\eda\sim_lib` directory.

The following is a VCS command for performing a functional RTL simulation with one of the libraries in Table 3–2:

`vcs -R` *<test bench>*`.v` *<design name>*`.v -v` *<library file>*`.v` ↵

Megafunctions Requiring Atom Libraries

The following Altera megafunctions require gate-level libraries to perform a functional simulation in a third-party simulator:

- altclkbuf
- altclkctrl
- altdq
- altdqs
- altddio_in
- altddio_out
- altddio_bidir
- altufm_none
- altufm_parallel
- altufm_spi
- altmemmult
- altremote_update

The gate-level library files are located in <*path to Quartus II installation*>**eda/sim_lib** directory (Table 3–3).

Functional RTL Simulation with Altera Memory Blocks

The VCS software supports functional simulation of complex Altera memory blocks such as lpm_ram_dp and altsyncram. You can create these memory blocks with the Quartus II MegaWizard® Plug-In Manager, which can be initialized with power-up data via a Hexadecimal (Intel-Format) File (**.hex**) or Memory Initialization File (**.mif**). The `lpm_file` parameter included in the file generated by the MegaWizard Plug-In Manager points to the path of the Hexadecimal (Intel-Format) File or Memory Initialization File that is used to initialize the memory block. You can create a Hexadecimal (Intel-Format) File or Memory Initialization File with the Quartus II software.

Compiling Functional Library Files with Compiler Directives

If you use a Hexadecimal (Intel-Format) File, no compiler directives are required. If you use a RAM Initialization File, the `USE_RIF` macro must be defined to compile the model library files. For example, enter the following when compiling the `altera_mf` library using RIF memory initialization files:

```
vcs -R -v <path to Quartus installation> /
\eda\sim_lib\altera_mf.v <test bench file> /
<design file (top-level)> +define+USE_RIF=1 ↵
```



For the Quartus II software versions 5.0 and earlier, the `NO_PLI` macro must be defined instead of `USE_RIF`. The `NO_PLI` macro is forward compatible with the Quartus II software.

Post-Synthesis Simulation

A post-synthesis simulation verifies the functionality of a design after synthesis has been performed. You can create a post-synthesis netlist in the Quartus II software and use this netlist to perform a post-synthesis simulation in the VCS software. Once the post-synthesis version of the design has been verified, the next step is to place-and-route the design in the target architecture using the Quartus II software.

Generating a Post-Synthesis Simulation Netlist

The following steps describe the process of generating a post-synthesis simulation netlist in the Quartus II software:

1. Perform Analysis and Synthesis. On the Processing menu, point to Start and click **Start Analysis and Synthesis**.
2. Turn on the **Generate Netlist for Functional Simulation Only** option by performing the following steps:
    a. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
    b. In the **Category** list, select **Simulation**. The **Simulation** page is shown.
    c. In the **Tool name** list, select **VCS**.
    d. You can modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
    e. Click **More Settings**. The **More EDA Tools Simulation Settings** dialog box appears. In the **Existing options settings list**, click **Generate Netlist for Functional Simulation Only** and select **On** from the **Setting** list under **Option**.
    f. Click **OK**.
    g. In the **Settings** dialog box, click **OK**.
3. Run the EDA Netlist Writer. On the Processing menu, point to Start and click **Start EDA Netlist Writer**.

During the EDA Netlist Writer stage, the Quartus II software produces a Verilog Output File (**.vo**) that can be used for the post-synthesis simulations in the VCS software. This netlist file is mapped to architecture-specific primitives. No timing information is included at this stage.

The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to the *<project directory>* **/simulation/vcs** directory. This netlist, along with the device family library listed in Table 3–3, can be used to perform a post-synthesis simulation in the VCS software.

Table 3–3. Altera Gate-Level Simulation Library Files

Library Files	Description
arriagx_atoms.v **arriagx_hssi_atoms.v**	Atom libraries for Arria™ GX designs
stratixii_atoms.v	Atom libraries for Stratix® II designs
stratixiigx_atoms.v **stratixiigx_hssi_atoms.v**	Atom libraries for Stratix II GX designs
stratix_atoms.v	Atom libraries for Stratix designs
stratixgx_atoms.v **stratixgx_hssi_atoms.v**	Atom libraries for Stratix GX designs
stratixiii_atoms.v	Atom libraries for Stratix III designs
hardcopyii_atoms.v	Atom libraries for HardCopy® II designs
hcstratix_atoms.v	Atom libraries for HardCopy Stratix designs
cycloneiii_atoms.v	Atom libraries for Cyclone® III designs
cycloneii_atoms.v	Atom libraries for Cyclone II designs
cyclone_atoms.v	Atom libraries for Cyclone designs
apexii_atoms.v	Atom libraries for APEX™ II designs
apex20ke_atoms.v	Atom libraries for APEX 20KE, APEX 20KC, and Excalibur™ designs
apex20k_atoms.v	Atom libraries for APEX 20K designs
flex10ke_atoms.v	Atom libraries for FLEX® 10KE and ACEX® 1K designs
flex6000_atoms.v	Atom libraries for FLEX 6000 designs
maxii_atoms.v	Atom libraries for MAX® II designs
max_atoms.v	Atom libraries for MAX 3000 and MAX 7000 designs
mercury_atoms.v	Atom libraries for Mercury™ designs

The following VCS software commands describe the command-line syntax used to perform a post-synthesis simulation with the appropriate device family library listed in Table 3–3:

```
vcs -R <test bench> <post synthesis netlist> -v <altera device family library> ↵
```

Gate-Level Timing Simulation

A gate-level timing simulation verifies the functionality of the design after place-and-route. You can create a post-fit netlist in the Quartus II software and use this netlist to perform a gate-level timing simulation in VCS software.

Generating a Gate-Level Timing Simulation Netlist

To perform gate-level timing simulation, the VCS software requires information about how the design was placed into device-specific architectural blocks. The Quartus II software provides this information in the form of a Verilog Output File for Verilog HDL designs. The accompanying timing information is stored in the Standard Delay Output File (**.sdo**), which annotates the delay for the elements found in the Verilog Output File.

The following steps describe the process of generating a gate-level timing simulation netlist in the Quartus II software:

1. Perform a full compilation. On the Processing menu, click **Start Compilation**.
2. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
3. In the **Category** list, select **Simulation**. The **Simulation** page is shown.
4. In the **Tool name** list, select **VCS**.
5. You can modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
6. Click **OK**.
7. In the **Settings** dialog box, click **OK**.
8. Run the EDA Netlist Writer. On the Processing menu, point to Start and click **Start EDA Netlist Writer**.

During the EDA Netlist Writer stage, the Quartus II software produces a Verilog Output File (**.vo**) that can be used for post-synthesis simulations in the VCS software. This netlist file is mapped to architecture-specific primitives. No timing information is included at this stage. The resulting netlist is located in the output directory you specified in the Settings dialog box, which defaults to the <*project directory*>**/simulation/vcs** directory.

Generating Different Timing Model

If you enable the Quartus II Classic or Quartus II TimeQuest Timing Analyzer when generating the SDO file, slow-corner (worst case) timing models are used by default. To generate the SDO file using a different timing model, you must run the Quartus II Classic or the Quartus II TimeQuest Timing Analyzer with a different timing model before you start the EDA Netlist writer.

To run the Classic Timing Analyzer with the best-case model, on the Processing menu, point to Start and click **Start Classic Timing Analyzer (Fast Timing Model)**. After timing analysis is complete, the Compilation Report appears. You can also type the following command at a command prompt:

```
quartus_tan <project_name> --fast_model=on ↵
```

To run the Quartus II TimeQuest Timing Analyzer with a best-case model, use the `-fast_model` option after you create the timing netlist. The following command enables the fast timing models:

```
create_timing_netlist -fast_model
```

You can also type the following command at a command prompt:

```
quartus_sta <project_name> --fast_model=on ↵
```



For more information about generating the timing model, refer to the *Quartus II Classic Timing Analyzer* or *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

After you run the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer, you can perform steps 2 through 8 in "Generating a Gate-Level Timing Simulation Netlist" on page 3–8 to generate the SDO file. For fast corner timing models, the **_fast** post fix is added to the VO, VHO, and SDO file (for example, **my_project_fast.vo**, **my_project_fast.vho**, and **my_project_fast.sdo**).

Operating Condition Example: Generate All Timing Models for Stratix III Devices

In Stratix III and Cyclone III devices, you can specify different temperature and voltage parameters to generate the timing models. Table 3–4 shows the available operation conditions (model, voltage, and temperature) for Stratix III and Cyclone III devices.

Table 3–4. Available Operating Condition for Stratix III and Cyclone III Devices

Device Family	Model	Voltage	Temperature
Stratix III	Slow	1100 mV	85° C
	Slow	1100 mV	0° C
	Fast	1100 mV	0° C
Cyclone III	Slow	1200 mV	85° C
	Slow	1200 mV	0° C
	Fast	1200 mV	0° C

To generating the SDO files for the three different operating conditions for a Stratix III design, perform the following steps:

1. Generate all the available corner models at all operating conditions. Type the following command at a command prompt:

```
quartus_sta <project name> --multicorner ↵
```

2. Generate the ModelSim simulation output files for all three corners specified above. The output files are generated in the simulation output directory. Type the following command at a command prompt:

```
quartus_eda <project name> --simulation --tool=vcs --format=verilog ↵
```

To summarize, for the three operating conditions the steps above generate the following files in the simulation output directory:

First slow corner (slow, 1100 mV, 85º C):
VO file— <*revision name*>**.vo**
SDO file— <*revision name*>**_v.sdo**

Second slow corner (slow, 1100 mV, 0º C):
VO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_slow.vo**
SDO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_v_slow.sdo**

Fast corner (fast, 1100 mV, 0º C):
VO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_fast.vo**
SDO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_v_fast.sdo**

Perform Timing Simulation Using Post-Synthesis Netlist

You can perform a timing simulation using the post-synthesis netlist instead of using a gate-level netlist and you can generate a SDO without running the fitter. In this case, the SDO file includes all timing values for the device cells only. Interconnect delays are not included because fitting (placement and routing) has not been performed.

To generate the post-synthesis netlist and the SDO file, type the following command at a command prompt:

```
quartus_map <project name> -c <revision name> ↵
quartus_tan <project name> -c <revision name> --post_map --zero_ic_delays ↵
quartus_eda <project name> -c <revision name> --simulation --tool=<third party EDA tool>
--format=<HDL language> ↵
```

For more information about the `-format` and `-tool` option, type the following command: `quartus_eda -help=<options> command`

Common VCS Software Compiler Options

The VCS software has options that help you simulate your design. Table 3–5 lists some of the options that are available.

Table 3–5. VCS Software Compiler Options (Part 1 of 2)

Library	Description
`-R`	Runs the executable file immediately.
`-RI`	Once the compile has completed, instructs the VCS software to automatically launch VirSim.
`-v` <*library filename*>	Specifies a Verilog HDL library file (for example, **220model.v** or **altera_mf.v**). The VCS software looks in this file for module definitions that are found in the source code. Only the relevant library files are compiled based on the modules found.
`-y` <*library directory*>	Specifies a Verilog HDL library directory. The VCS software looks for library files in this folder that contain module definitions that are instantiated in the source code.

Table 3–5. VCS Software Compiler Options (Part 2 of 2)

Library	Description
`+compsdf`	Indicates that the VCS software compiler includes the back-annotated SDF file in the compilation.
`+cli`	The VCS software enters Command-Line Interface (CLI) mode upon successful compilation completion.
`+race`	Specifies that the VCS software generate a report that indicates all of the race conditions in the design. Default report name is **race.out**.
`-P`	Compiles user-defined Programming Language Interface (PLI) table files.
`-q`	Indicates the VCS software runs in quiet mode. All messages are suppressed.



For more information about any VCS software option, refer to the *VCS User Guide.*

Using VirSim

VirSim is the graphical debugging system for the VCS software. This tool is included with the VCS software and can be run by using the `-RI` compile-time compiler option when compiling a design. The following VCS software command describes the command-line syntax for compiling and loading a timing simulation in VirSim:

```
vcs -RI <test bench>.v <design name>.vo -v <path to Quartus II installation> \
\eda\sim_lib\<device family>_atoms.v +compsdf ↵
```



For more information about using VirSim, refer to the *VirSim User Manual* included in the VCS software installation.

Debugging Support Command-Line Interface

The VCS software has an interactive non-graphical debugging capability that is very similar to other UNIX debuggers such as the GNU debugger (GDB). The VCS software CLI can be used to halt simulations at user-defined break points, force registers with values, and display values of registers.

Enable the non-graphical capability by using the `+cli` run-time option. Use the VCS software CLI to debug your Altera FPGA design by typing the following command:

```
vcs -R <test bench>.v <design name>.vo
    -v <path to Quartus II installation> \
    \eda\sim_lib\<device family>_atoms.v +compsdf +cli ↵
```

The +cli command takes an optional number argument that specifies the level of debugging capability. As the optional debugging capability is increased, the overhead incurred by the simulation is increased, resulting in an increase in simulation times.

For more information about the +cli options, refer to the *VCS User Guide* included in the VCS software installation.

For the design examples to run gate-level timing simulation in VHDL or Verilog language, refer to **www.altera.com/support/examples/vcs/exm-vcs.html**.

Simulating Designs that Include Transceivers

If your design includes a Stratix GX or Stratix II GX transceiver, you must compile additional library files to perform functional or timing simulations.

Stratix GX Functional Simulation

To perform a functional simulation of your design that instantiates the altgxb megafunction, enabling the gigabit transceiver block gigabit transceiver block on Stratix GX devices, compile the **stratixgx_mf** model file into the **altgxb** library.

The **stratixiigx_mf** model file references the **lpm** and **sgate** libraries, so you must create these libraries to perform a simulation.

Example of Compiling Library Files for Functional Stratix GX Simulation in Verilog HDL

To compile the libraries necessary for functional simulation of a Verilog HDL design targeting a Stratix GX device, type the following commands at the VCS command prompt:

```
vcs -R <test bench>.v <design files>.v -v stratixgx_mf.v -v \
sgate.v -v 220model.v -v altera_mf.v ↵
```

Stratix GX Post-Fit (Timing) Simulation

Perform a post-fit timing simulation of your design that includes a Stratix GX transceiver by compiling the **stratixgx_atoms** and **stratixgx_hssi_atoms** model files into the **stratixgx** and **stratixgx_gxb** libraries, respectively.



The **stratixgx_hssi_atoms** model file references the **lpm** and **sgate** libraries, so you must create these libraries to perform a simulation.

Example of Compiling Library Files for Timing Stratix GX Simulation in Verilog HDL

To compile the libraries necessary for timing simulation of a Verilog HDL design targeting a Stratix GX device, type the following commands at the VCS command prompt:

```
vcs -R <testbench>.v <gate-level netlist>.vo -v stratixgx_atoms.v -v \
stratixgx_hssi_atoms.v -v sgate.v -v 220model.v -v altera_mf.v \
+transport_int_delays +pulse_int_e/0 +pulse_int_r/0 \
+transport_path_delays +pulse_e/0 +pulse_r/0 ↵
```

Stratix II GX Functional Simulation

To perform a functional simulation of your design that instantiates the alt2gxb megafunction, enabling the gigabit transceiver block gigabit transceiver block on Stratix II GX devices, compile the **stratixiigx_hssi** model file into the **stratixiigx_hssi** library.



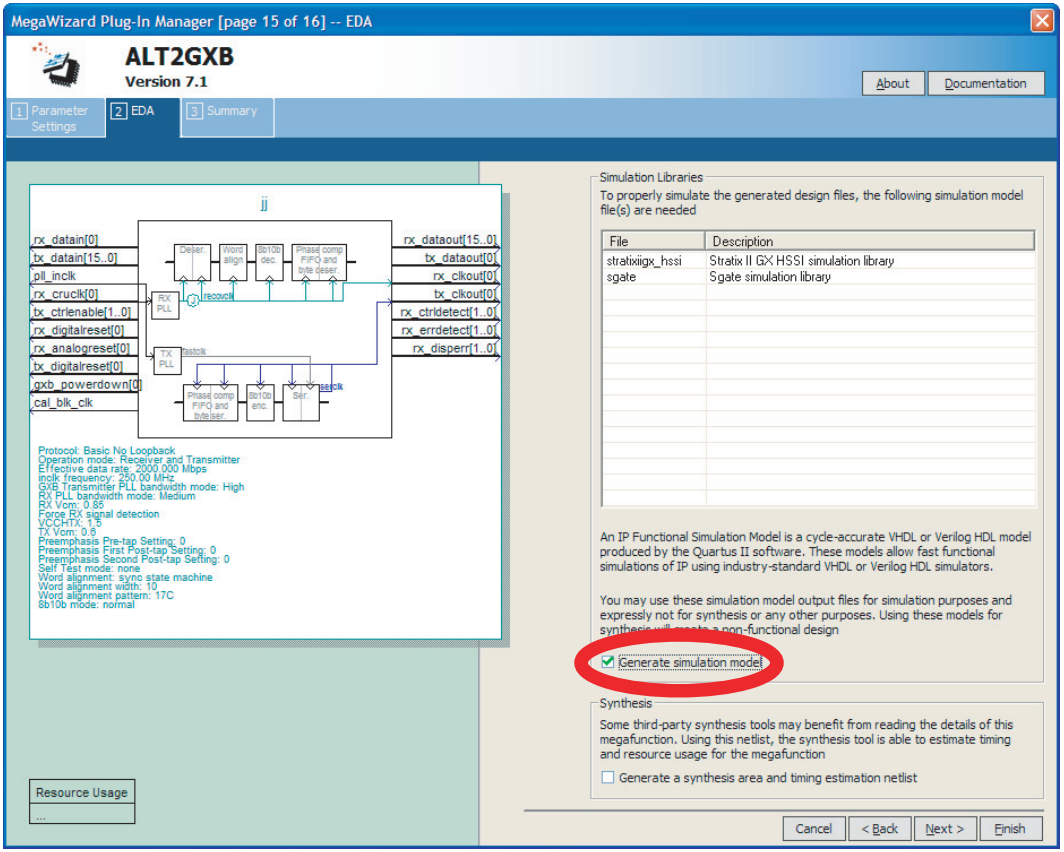
The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries, so you must create these libraries to perform a simulation.

Generate a functional simulation netlist by turning on **Generate Simulation Model in the Simulation Library** in the alt2gxb MegaWizard Plug-In Manager (Figure 3–2). The *<alt2gxb entity name>***.vho** file or *<alt2gxb module name>***.vo** file is generated in the current project directory.



The Quartus II-generated `alt2gxb` functional simulation library file references `stratixiigx_hssi` wysiwyg atoms.

Figure 3–2. alt2gxb MegaWizard



Example of Compiling Library Files for Functional Stratix II GX Simulation in Verilog HDL

To compile the libraries necessary for functional simulation of a Verilog HDL design targeting a Stratix II GX device, type the following commands at the VCS command prompt:

```
vcs -R <testbench>.v <alt2gxb simulation netlist>.vo -v stratixgx_hssi_atoms.v -v \
sgate.v -v 220model.v -v altera_mf.v ↵
```

Stratix II GX Post-Fit (Timing) Simulation

To perform a post-fit timing simulation of your design that includes a Stratix II GX transceiver, compile `stratixiigx_atoms` and `stratixiigx_hssi_atoms` into the **stratixiigx** and **stratixiigx_hssi** libraries, respectively.



The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries, so you must create these libraries to perform a simulation.

Example of Compiling Library Files for Timing Stratix II GX Simulation in Verilog HDL

To compile the libraries necessary for timing simulation of a Verilog HDL design targeting a Stratix II GX device, type the following commands at the VCS command prompt:

```
vcs -R <testbench>.v <gate-level netlist>.vo -v stratixiigx_atoms.v -v \
stratixiigx_hssi_atoms.v -v sgate.v -v 220model.v -v altera_mf.v \
+transport_int_delays +pulse_int_e/0 +pulse_int_r/0 \
+transport_path_delays +pulse_e/0 +pulse_r/0 ↵
```

Using PLI Routines with the VCS Software

The VCS software can interface your custom-defined C code with Verilog HDL source code. This interface is known as PLI. This interface is extremely useful because it allows advanced users to define their own system tasks that currently may not exist in the Verilog HDL.

Preparing and Linking C Programs to Verilog HDL Code

When compiling the source code, the C code must include a reference to the **vcsuser.h** file. This file defines PLI constants, data structures, and routines that are necessary for the PLI interface. This file is included with the VCS software installation and can be found in the **$VCS_HOME\lib** directory.

Once the C code is complete, you must create an object file (**.o**). Create the object file with the following command:

```
gcc -c my_custom_function.c ↵
```

Next, you must create a PLI table file (**.tab**). This file maps the C program task to the matching task `$task` in the Verilog HDL source code. You can create this file using a standard text editor. The following is an example of an entry in the PLI file:

```
$my_custom_function call=my_custom_function acc+=rw* ↵
```

The Verilog HDL code can now include a reference to the user-defined task. To compile an Altera FPGA design that includes a reference to a user-defined system task, type the following at the command-line prompt:

```
vcs -R <test bench>.v <design name>.v -v <Altera library file>.v -P <my_tabfile.tab> \
<my_custom_function.o> ↵
```

Transport Delays

By default, the VCS software filters out all pulses that are shorter than the propagation delay between primitives. Turning on the transport delay options in the VCS software prevents the simulation tool from filtering out these pulses. Use the following options to ensure that all signal pulses are seen in the simulation results.

+transport_path_delays

Use this option when the pulses in your simulation may be shorter than the delay within a gate-level primitive. For this option to work you must also include the `+pulse_e/number` and `+pulse_r/number` options.

+transport_int_delays

Use this option when the pulses in your simulation may be shorter than the interconnect delay between gate-level primitives. For this option to work, you must also include the `+pulse_int_e/number` and `+pulse_int_r/number` options. The `+transport_path_delays` and `+transport_int_delays` options are also used by default in the NativeLink feature for gate-level timing simulation.



For more information about either of these options, refer to the *VCS User Guide* installed with the VCS software.

The following VCS software command describes the command-line syntax to perform a post-synthesis simulation with the device family library:

```
vcs -R <test bench>.v <gate-level netlist>.v -v <altera device family library>.v \
+transport_int_delays +pulse_int_e/0 +pulse_int_r/0 \
+transport_path_delays +pulse_e/0 +pulse_r/0 ↵
```

Using NativeLink with the VCS Software

The NativeLink® feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools and allows you to run VCS within the Quartus II software.

Setting Up NativeLink

To run VCS automatically from the Quartus II software using the NativeLink feature, you must specify the path to your simulation tool by performing the following steps:

1. On the Tools menu, click **Options.** The **Options** dialog box appears.
2. In the **Category** list, select **EDA Tool Options**. The **EDA Tool Options** page is shown.
3. Double-click the entry under the **Location of executable** column.
4. Type or browse to the directory containing the executables of your EDA tool.
5. Click **OK**.

You can also specify the path to the simulator's executables by using the `set_user_option` Tcl command:

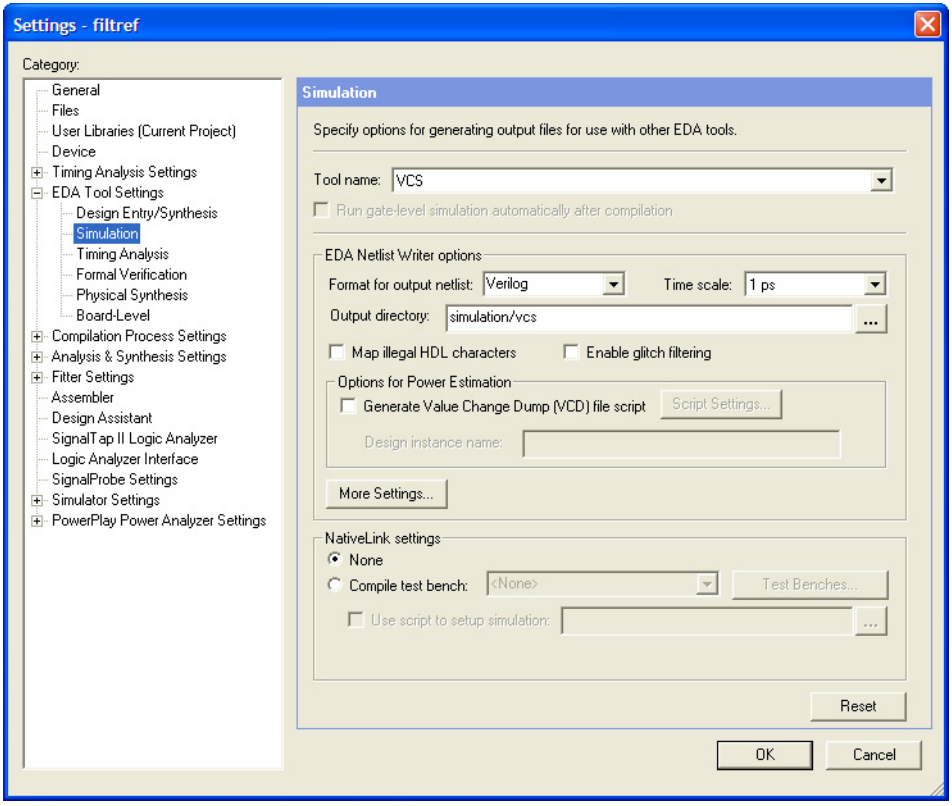
```
set_user_option -name EDA_TOOL_PATH_VCS <path to executables> ↵
```

Performing an RTL Simulation Using NativeLink

To run a functional RTL simulation automatically with the VCS software in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page is shown (Figure 3–3).

Figure 3–3. Simulation Page in the Settings Dialog Box



3. In the **Tool name** list, select **VCS**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.

4. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.

For more information about setting up a testbench with NativeLink, refer to "Setting Up a Testbench" on page 3–21.

5. Click **OK.**

6. On the Processing menu, point to Start and click **Start Analysis and Elaboration** to perform an analysis and elaboration. This command collects all your file name information and builds your design hierarchy in preparation for simulation.

7. On the Tools menu, point to **EDA Simulation Tool** and click **Run EDA RTL Simulation** to automatically launch VCS, compile all necessary design files, and complete a simulation.

Performing a Gate-Level Simulation Using NativeLink

To run a gate-level timing simulation with the VCS software automatically in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.

2. In the **Category** list, select **Simulation**. The **Simulation** page is shown (Figure 3–3).

3. In the **Tool name** list, select **VCS**.

4. You can modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.

5. To perform a gate level simulation after each full compilation, turn on **Run Gate Level Simulation automatically after compilation**.

6. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.

7. Click **OK**.

8. Perform a full compilation. On the Processing menu, click **Start Compilation**.

9. On the Processing menu, point to Start and click **Start EDA Netlist Writer** to generate a simulation netlist of your design.

10. On the Tools menu, point to EDA Simulation Tool and click **Run EDA Gate Level Simulation** to automatically launch VCS, compile all necessary design files, and complete a simulation.



A VCS file (***.vcs**) is generated in the <*project_directory*>**\simulation\vcs** directory while running the NativeLink. With this VCS File (***.vcvs**), you can simulating the design using the following command without using the NativeLink:

```
vcs -file <project_directory>\simulation\vcs\<generated_do_file>.vcs
```

Setting Up a Testbench

You can automatically launch your EDA simulator tool, compile your design files and testbench files, and perform a simulation automatically using the NativeLink feature.

To setup NativeLink with a testbench, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the "+" icon to expand **EDA Tool Settings** and select **Simulation**. The **Simulation** page is shown.
3. Under **NativeLink settings**, select **None** or **Compile test bench** (Table 3–6).

Table 3–6. NativeLink Settings

Settings	Description
None	Compile simulation models and design files.
Compile test bench	NativeLink compiles simulation models, design files, testbench files, and starts simulation.

4. If you select **Compile test bench**, select your testbench setup from the **Compile test bench** list. You can use different testbench setups to specify different testbench files for different test scenarios. If there are no testbench setups entered, create a testbench setup by performing the following steps:
    a. Click **Test Benches**. The **Test Benches** dialog box appears.
    b. Click **New**. The **New Test Bench Settings** dialog box appears.
    c. In the **Test Bench name** box, type in the testbench setup name which is used to identify between the different testbench setups.

d. In the **Test bench entity** box, type in the top-level entity name. For example, for a Quartus II generated Verilog testbench, type in <*Vector Waveform File name*>`_vlg_vec_tst`.

e. In the **Instance** box, type the full instance path to the top level of your FPGA design. For example, for a Quartus II generated Verilog test bench, type `i1`.

f. Under **Simulation period**, select **Run simulation until all vector stimuli are used**. If you select **End simulation at**, specify the simulation end time and the time unit.

g. Under **Test bench files**, browse and add all your testbench files in the **File name** box. Use the **Up** and **Down** buttons to reorder your files. The script used by NativeLink compiles the files in the order from top to the bottom.



You can also specify the library name and the HDL version to compile the testbench file. NativeLink compiles the testbench to the library name of the HDL specified version.

h. Click **OK**.

i. In the **Test Benches** dialog box, click **OK**.

Creating a Testbench

In the Quartus II software, you can create a Verilog HDL or VHDL testbench from a Vector Waveform File. The generated testbench includes the behavior of the input stimulus and applies it to your instantiated top-level FPGA design.

1. On the File menu, click **Open**. The **Open** dialog box appears.
2. Click the **Files of type** arrow and select **Waveform/Vector Files**. Select your file.
3. Click **Open**.
4. On the File menu, click **Export**. The **Export** dialog box appears.
5. Click the **Save as type** arrow and select **VHDL Test Bench File (*.vht)** or **Verilog Test Bench File (*.vt)**.
6. You can turn on **Add self-checking code to file** to check your simulation results against your Vector Waveform File.

7. Click **Export**.

Scripting Support

You can run procedures and create settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For more information about command-line scripting, refer to the *Command Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For detailed information about scripting command options, refer to the **Qhelp** utility.

To start the **Qhelp** utility, type this command:

```
quartus_sh --qhelp ↵
```

Generating a Post-Synthesis Simulation Netlist for VCS

You can use the Quartus II software to generate a post-synthesis simulation netlist with Tcl commands or with a command at a command prompt.

Tcl Commands

Use the following Tcl commands:

```
set_global_assignment -name EDA_SIMULATION_TOOL "VCS" ↵
set_global_assignment -name EDA_GENERATE_FUNCTIONAL_NETLIST ON ↵
```

Command Prompt

Use the following command to generate a simulation output file for the VCS software simulator; specify VHDL or Verilog HDL for the format:

```
quartus_eda <project name> --simulation=on --format=<format> --tool=vcs
--functional ↵
```

Generating a Gate-Level Timing Simulation Netlist for VCS

You can use the Quartus II software to generate a gate-level timing simulation netlist with Tcl commands or with a command at a command prompt.

Tcl Commands

Use the following Tcl commands:

```
set_global_assignment -name EDA_SIMULATION_TOOL "VCS" ↵
```

Command Prompt

Use the following command to generate a simulation output file for the VCS software simulator. Specify VHDL or Verilog HDL for the format.

```
quartus_eda <project name> --simulation=on --format=<format> --tool=vcs ↵
```

Conclusion

You can use the VCS software in your Altera FPGA design flow to easily and accurately perform functional RTL simulations, post-synthesis simulations, and gate-level functional timing simulations. The seamless integration of the Quartus II software and VCS software make this simulation flow an ideal method for fully verifying an FPGA design.

Referenced Documents

This chapter references the following documents:

- *Quartus II Installation and Licensing for Windows* Manual
- *Quartus II Installation and Licensing for UNIX and Linux Workstation* Manual
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *VCS User Guide*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Command Line Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 3–7 shows the revision history for this chapter.

Table 3–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Updated Table 3–1. • Updated "Operating Condition Example: Generate All Timing Models for Stratix III Devices" on page 3–10.	Updated for the Quartus II software version 7.2.
May 2007 v7.1.0	• Updated Tables 3–1, 3–2, and 3–3. • Updated "Generating a Gate-Level Timing Simulation Netlist" on page 3–8. • Added "Perform Timing Simulation Using Post-Synthesis Netlist" on page 3–11. • Updated "Performing a Gate-Level Simulation Using NativeLink" on page 3–20. • Updated procedures in "Setting Up a Testbench" on page 3–21.	Updated for the Quartus II software version 7.1.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only.	—
November 2006 v6.1.0	• Updated for the Quartus II software version 6.1. • Added library for Stratix III support. • Minor updates to Table 3-1, 3-2, and 3-3.	Updated for the Quartus II software version 6.1.
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0: • Added a section on setting VCS as the Simulation Tool • Updated EDA Tools Settings in the GUI. • Updated the Synopsys Design Constraints File information. • Added pulse_e and pulse_r information to simulation sections. • Added Quartus II-Generated Testbench information • Updated megafunction information.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
May 2005 v5.0.0	• Updated information. • Updated tables. • Added Using NativeLink® with VCS section. • New functionality for Quartus II software version 5.0.	
December 2004 v2.1	• Updates to tables, figures. • New functionality for Quartus II software version 4.2.	
June 2004 v2.0	• Updates to tables and figures. • New functionality for the Quartus II software version 4.1.	
February 2004 v1.0	• Initial release.	

4. Cadence NC-Sim Support

QII53003-7.2.0

Introduction

This chapter is a getting started guide to using the Cadence Incisive verification platform software in Altera® FPGA design flows. The Incisive verification platform software includes NC-Sim, NC-Verilog, NC-VHDL, Verilog, and VHDL desktop simulators. This chapter provides step-by-step explanations of the basic NC-Sim, NC-Verilog, and NC-VHDL functional, post-synthesis, and gate-level timing simulations. It also describes the location of the simulation libraries and how to automate simulations.

This chapter contains the following topics:

- "Software Requirements"
- "Functional and RTL Simulation" on page 4–6
- "Post-Synthesis Simulation" on page 4–22
- "Gate-Level Timing Simulation" on page 4–24
- "Simulating Designs that Include Transceivers" on page 4–31
- "Using the NativeLink Feature with NC-Sim" on page 4–37
- "Incorporating PLI Routines" on page 4–43
- "Scripting Support" on page 4–48

Software Requirements

You must first install the Quartus® II software before using it with the Cadence Incisive verification platform software. The Cadence interface is installed automatically when you install the Quartus II software on your computer.

Table 4–1 shows the Cadence NC simulator versions compatible with specific Quartus II software versions.

Table 4–1. Compatibility Between Software Versions

Quartus II Software	Cadence NC Simulators (UNIX)	Cadence NC Simulators (PC)	Cadence NC Simulators (Linux)
Version 7.2	Version 6.10 p001	Version 5.4 s011	Version 6.10 p001
Version 7.1	Version 5.83 p003	Version 5.4 s011	Version 5.83 p003
Version 6.1 and 7.0	Version 5.82 p001	Version 5.4 s011	Version 5.82 p001
Version 6.0	Version 5.5 s012	Version 5.4 s011	Version 5.5 s012
Version 5.1	Version 5.4 s011	Version 5.4 s011	Version 5.4 s011
Version 5.0	Version 5.4 s004	Version 5.4 p001	Version 5.4 s004
Version 4.2	Version 5.1 s017	Version 5.1 s017	Version 5.1 s017
Version 4.1	Version 5.1 s012	Version 5.1 s010	Version 5.0 p001
Version 4.0	Version 5.0 s005	Version 5.0 s006	Version 5.0 p001

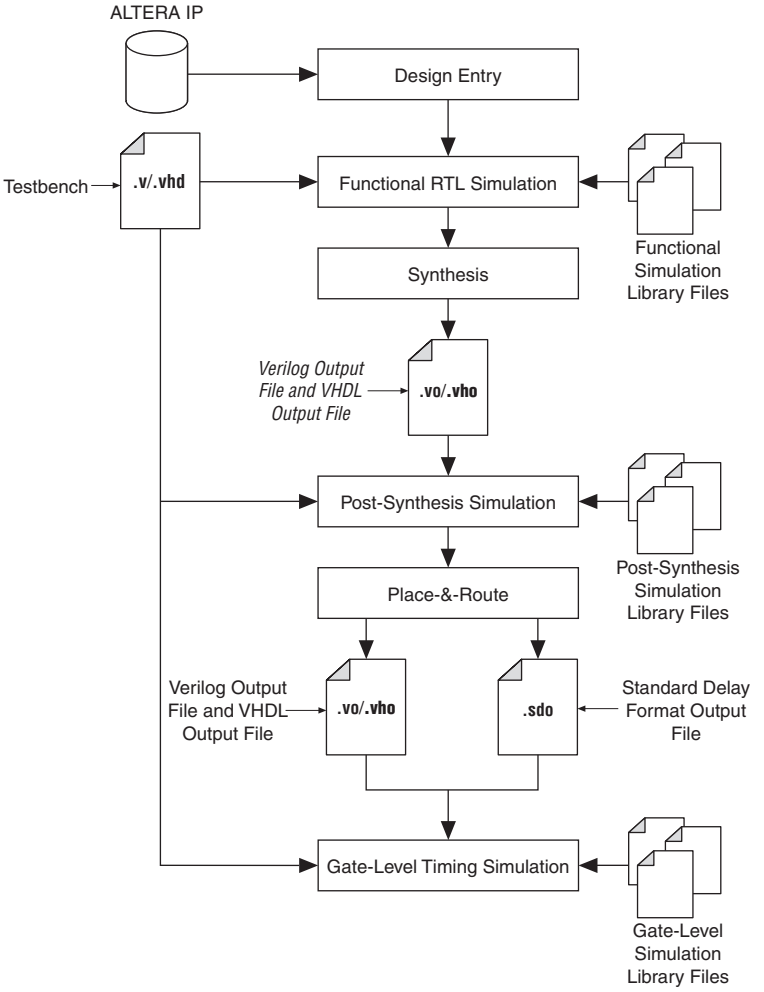
Simulation Flow Overview

The Incisive platform software supports the following simulation flows:

- Functional and RTL Simulation
- Post-Synthesis Simulation
- Gate-Level Timing Simulation
- Using the NativeLink Feature with NC-Sim

Figure 4–1 shows the Quartus II software and Cadence design flow.

Figure 4–1. Quartus II Software Design Flow with Cadence NC Simulators



Functional and RTL simulation verifies the functionality of your design. When you perform a functional simulation with Cadence Incisive simulators, you use your design files (Verilog HDL or VHDL) and the models provided with the Quartus II software. These Quartus II models are required if your design uses the library of parameterized modules (LPM) functions or Altera-specific megafunctions. Refer to "Functional and RTL Simulation" on page 4–6 for more information about how to perform this simulation.

A post-synthesis simulation verifies the functionality of a design after synthesis has been performed. You can create a post-synthesis netlist (**.vo** or **.vho**) in the Quartus II software and use this netlist to perform a post-synthesis simulation with the Incisive simulator. Refer to "Post-Synthesis Simulation" on page 4–22 for more information about how to perform this simulation.

After performing place-and-route, the Quartus II software generates a Verilog Output File (**.vo**) or VHDL Output File (**.vho**) and a Standard Delay Output file (**.sdo**) for gate-level timing simulation. The netlist files map your design to architecture-specific primitives. The SDO file contains the delay information of each architecture primitive and routing element specific to your design. Together, these files provide an accurate simulation of your design with the selected Altera FPGA architecture. Refer to "Gate-Level Timing Simulation" on page 4–24 for more information about how to perform this simulation.

Operation Modes

In the NC simulators, you can use either the GUI mode or the command-line mode to simulate your design.

You can start the Incisive simulators in GUI mode in a PC or a UNIX environment by typing `nclaunch` at a command prompt.

To simulate in command-line mode, use the programs shown in Table 4–2.

Table 4–2. Command-Line Programs

Program	Function
ncvlog or **ncvhdl**	NC-Verilog (**ncvlog**) compiles your Verilog HDL code into a Verilog Syntax Tree (**.vst**) file. **ncvlog** also performs syntax and static semantics checks. NC-VHDL (**ncvhdl**) compiles your VHDL code into a VHDL Syntax Tree (**.ast**) file. **ncvhdl** also performs syntax and static semantics checks.
ncelab	NC-Elab (**ncelab**) elaborates the design. **ncelab** constructs the design hierarchy and establishes signal connectivity. This program also generates a Signature File (**.sig**) and a Simulation SnapShot File (**.sss**).
ncsim	NC-Sim (**ncsim**) performs mixed-language simulation. This program is the simulation kernel that performs event scheduling and executes the simulation code.

Quartus II Software and NC Simulation Flow Overview

An overview of the Quartus II software and Cadence NC simulation flow is described below. More detailed information is provided in "Functional and RTL Simulation" on page 4–6, "Post-Synthesis Simulation" on page 4–22, and "Gate-Level Timing Simulation" on page 4–24.

1. Set up your working environment (UNIX only).

   You must set several environment variables in UNIX to establish an environment that facilitates entering and processing designs.

2. Create user libraries.

   Create a file that maps logical library names to their physical locations. These library mappings include your working directory and any design-specific libraries; for example, Altera LPM functions or megafunctions.

3. Compile source code and testbenches.

   You compile your design files at the command-line using **ncvlog** (Verilog HDL files) or **ncvhdl** (VHDL files), or on the Tools menu by clicking **Verilog Compiler** or **VHDL Compiler** in NCLaunch. During compilation, the NC software performs syntax and static semantic checks. If no errors are found, compilation produces an

internal representation for each HDL design unit in the source files. By default, these intermediate objects are stored in a single, packed, library database file in your working directory.

4. Elaborate your design.

   Before you can simulate your model, you must define the design hierarchy in a process called elaboration. Use **ncelab** in command-line mode or on the Tools menu, click **Elaborator** in NCLaunch to elaborate the design.

5. Add signals to your waveform.

   Before simulating, specify which signals to view in your waveform using a simulation history manager (SHM) database.

6. Simulate your design.

   Run the simulator with the **ncsim** program (command-line mode) or by clicking **Run** in the SimVision Console window.

Functional and RTL Simulation

The following sections provide detailed instructions for performing functional/RTL simulation using the Quartus II software and the Cadence Incisive platform software tools.

Create Libraries

Before simulating with the Incisive simulator, you must set up libraries with a file named **cds.lib**. The **cds.lib** file is an ASCII text file that maps logical library names—for example, your working directory or the location of resource libraries such as models for LPM functions—to their physical directory paths. When you run the Incisive simulator, the tool reads **cds.lib** to determine which libraries are accessible and where they are located. There is also a default **cds.lib** file, which you can modify for your project settings.

You can use more than one **cds.lib** file. For example, you can have a project-wide **cds.lib** file that contains library settings specific to a project such as technology or cell libraries and a user **cds.lib** file.

The following sections describe how to create and edit a **cds.lib** file:

- Basic libraries setup
- LPM function, Altera megafunction, and Altera primitive library setup

Basic Library Setup

You can create a **cds.lib** file with any text editor. The following examples show how you use the DEFINE statement to bind a library name to its physical location. The logical and physical names can be the same or you can select different names. The DEFINE statement usage is:

DEFINE <*library name*> <*physical directory path*>

For example, a simple **cds.lib** file for Verilog HDL contains the following lines:

```
DEFINE lib_std /usr1/libs/std_lib
DEFINE worklib ../worklib
```

Using Multiple cds.lib Files

Use the INCLUDE or SOFTINCLUDE statement to reference another **cds.lib** file within a **cds.lib** file. The syntax is:

INCLUDE <*path to another cds.lib*>

or

SOFTINCLUDE <*path to another cds.lib*>



For the Windows operating system, enclose the path with quotation marks if there are spaces in the directory path.

For VHDL or mixed-language simulation, in addition to the DEFINE statements, you must include the default **cds.lib** file (included with NC-Sim). The syntax for including the default **cds.lib** file is:

INCLUDE <*path to NC installation*>/tools/inca/files/cds.lib

or

INCLUDE $CDS_INST_DIR/tools/inca/files/cds.lib

The default **cds.lib** file, provided with NC tools, contains a SOFTINCLUDE statement to include other **cds.lib** files, such as **cdsvhdl.lib** and **cdsvlog.lib**. These files contain library definitions for IEEE libraries and Synopsys libraries.

Create a cds.lib File in Command-Line Mode

To create a **cds.lib** file at a the command prompt, perform the following steps:

1. Create a directory for the work library and any other libraries you need by typing the following command at a command prompt:

   `mkdir <library name>` ↵

   For example: `mkdir worklib` ↵

2. Using a text editor, create a **cds.lib** file and add the following line to it:

   `DEFINE <library name> <physical directory path>`

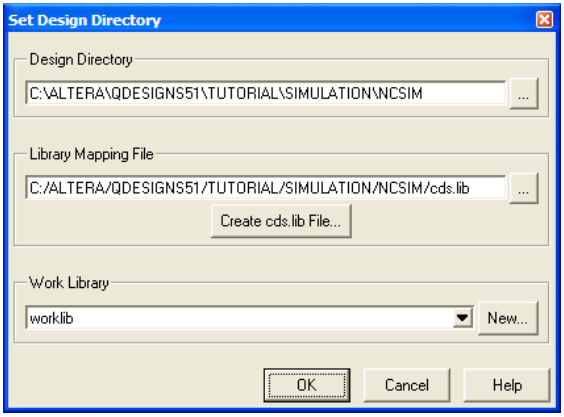
   For example: `DEFINE worklib ./worklib`

Create a cds.lib File in GUI Mode

To create a **cds.lib** file using the GUI, perform the following steps:

1. Type `nclaunch` ↵ at the command line to run the GUI.
2. If the NCLaunch window is not in multiple step mode, on the File menu, click **Switch to Multiple Step**.
3. Change your design directory on the File menu by clicking **Set Design Directory**. The **Set Design Directory** dialog box appears (Figure 4–2).

Figure 4–2. Creating a Work Directory in GUI Mode



4. Click **Browse** to navigate to your design directory.
5. Click **Create cds.lib File** and in the **New cds.lib File** dialog box, click **Save** and choose the libraries you want to include.
6. Click **New** under **Work Library**.
7. Enter your new work library name, for example, `worklib`.
8. Click **OK**. The new library is displayed under **Work Library**. Figure 4–2 shows an example using the directory name **worklib**.
9. Repeat steps 7 and 8 for each functional simulation library. For example, `lpm`, `altera_mf`, and `altera`.
10. Click **OK** in the **Set Design Directory** dialog box.



You can edit your libraries by editing the **cds.lib** file. Edit the **cds.lib** file by right-clicking the **cds.lib** filename in the right side of the window and choosing **Edit**.

LPM Functions, Altera Megafunctions, and Altera Primitives Libraries

Altera provides behavioral descriptions for LPM functions, Altera-specific megafunctions, and Altera primitives. You can implement the megafunctions in a design using the Quartus II MegaWizard® Plug-In Manager or by instantiating them directly from your design file. You must set up resource libraries so that you can simulate your design in the Incisive simulator if your design uses LPM functions, Altera megafunctions, or Altera primitives.



Many LPM functions and Altera megafunctions use memory files. You must convert the memory files into a format the Incisive tools can read before simulating. Follow the instructions in "Compile Source Code and Testbenches" on page 4–13 to connect the memory files.

Altera provides megafunction behavioral descriptions in the files shown in Table 4–3. These library files are located in the following directory:

<path to Quartus II installation>/**eda**/**sim_lib** directory.

For more information about LPM functions and Altera megafunctions, refer to the Quartus II Help.

Table 4–3. Megafunction Behavioral Description Files

Library Description	Verilog HDL	VHDL
LPM	**220model.v**	**220model.vhd** *(1)* **220model_87.vhd** *(2)* **220pack.vhd**
Altera megafunction	**altera_mf.v**	**altera_mf.vhd** *(1)* **altera_mf_87.vhd** *(2)* **altera_components.vhd**
Altera primitives	**altera_primitives.v**	**altera_primitives.vhd** *(1)* **altera_primitives_components.vhd**
IP functional simulation model	**sgate.v**	**sgate.vhd** **sgate_pack.vhd**
`altgxb`	**stratixgx_mf.v** *(3)*	**stratixgx_mf.vhd** *(3)* **stratixgx_mf_components.vhd** *(3)*
`alt2gxb`	**stratixiigx_hssi_atoms.v,** **arriagx_hssi_atoms.v***(3)*, *(4)*	**stratixiigx_hssi_atoms.vhd** **stratixiigx_hssi_components.vhd** **arriagx_hssi_atoms.vhd** **arriagx_hssi_components.vhd** *(3)*, *(4)*

Notes to Table 4–3:
(1) Use this model with VHDL 93.
(2) Use this model with VHDL 87.
(3) The `alt2gxb` and `altgxb` library files require the **lpm** and **sgate** libraries.
(4) You must generate a functional simulation netlist for simulation.

If an `lpm` library does not exist, set up a library for LPM functions by creating a new directory and adding the following line to your **cds.lib** file:

```
DEFINE lpm <path>/<directory name>
```

If an `altera_mf` library does not exist, set up a library for Altera megafunctions by adding the following line to your **cds.lib** file:

```
DEFINE altera_mf <path>/<directory name>
```

Megafunctions Requiring Atom Libraries

The following Altera megafunctions require device atom libraries to perform a functional simulation in a third-party simulator:

- altclkbuf
- altclkctrl
- altdq
- altdqs
- altddio_in
- altddio_out
- altddio_bidir
- altufm_none
- altufm_parallel
- altufm_spi
- altmemmult
- altremote_update

The device atom library files are located in the following directory:

*<path to Quartus II installation>***/eda/sim_lib**

Simulating a Design with Memory

The NC-Sim simulator supports simulating Altera memory megafunctions initialized with Hexadecimal (Intel-Format) File (**.hex**) or RAM Initialization Files (**.rif**).

Although synthesis is able to read a Memory Initialization File (**.mif**), these files are not supported in simulations with third-party tools and must be converted to either a Hexadecimal (Intel-Format) File or RAM Initialization File.

Table 4–4 summarizes the different types of memory initialization file formats that are supported with each RTL language.

Table 4–4. Simulation Support for Memory Initialization Files

File	Verilog HDL	VHDL
Hexadecimal (Intel-Format) File	Yes *(1)*	Yes
Memory Initialization File	No	No
RAM Initialization File	Yes *(2)*	No

Notes to Table 4–4:

(1) For memories and library files from Quartus II software version 5.0 and earlier, you are required to use a `PLI` library containing the `convert_hex2ver` task function.

(2) Requires the `USE_RIF` macro to be defined.

To convert your Memory Initialization File into either a Hexadecimal (Intel-Format) File or RAM Initialization File, perform the following steps:

1. Open the Memory Initialization File and on the File menu, click **Export**. The **Export** dialog box appears.

2. Select **Hexadecimal (Intel-Format) File (*.hex)** or **RAM Initialization File (*.rif)** from the **Save as type** list and click **OK**.



Alternatively, you can convert a Memory Initialization File to a RAM Initialization File using the `mif2rif.exe` executable located in the <*Quartus II installation*>**/bin** directory. An example of this executable is:

```
mif2rif <mif_file> <rif_file>
```

3. Modify the HDL file generated with the MegaWizard Plug-In Manager.

   The MegaWizard Plug-In Manager-generated Altera memory megafunction wrapper file includes the `lpm_file` parameter for LPM memories, or the `init_file` parameter for Altera-specific memories to point to the initialization file.

   In a text editor, open the MegaWizard Plug-In Manager generated wrapper file and edit the `lpm_file` or `init_file` parameters to point to the Hexadecimal (Intel-Format) File or RAM Initialization File, as shown below:

   ```
   lpm_ram_dp_component.lpm_file = "<path to HEX/RIF>"
   ```

4. Compile the functional library files with compiler directives.

   If you use a Hexadecimal (Intel-Format) File, no compiler directives are required. If you use a RAM Initialization File, the `USE_RIF` macro must be defined when compiling the model library files. For example, the following should be entered when compiling the `altera_mf` library when RAM Initialization Files are used:

   ```
   ncvlog -work altera_mf altera_mf.v -DEFINE
   "USE_RIF=1"
   ```



For Quartus II software versions 5.0 and earlier, you must define the `NO_PLI` macro instead of `USE_RIF`. The `NO_PLI` macro is forward compatible with the Quartus II software.

Compile Source Code and Testbenches

Compile your testbench and design files with **ncvlog** (for Verilog HDL files) and **ncvhdl** (for VHDL files). Both **ncvlog** and **ncvhdl** perform syntax checks and static semantic checks. A successful compilation produces an internal representation for each HDL design unit in the source files. By default, these intermediate objects are stored in a single, packed, library database file in your work library directory.

Compilation in Command-Line Mode

To compile from the command line, use one of the following commands:



You must create a work library before compiling.

- Verilog HDL:

```
ncvlog <options> -work <library name> <design files> ↵
```

- VHDL:

```
ncvhdl <options> -work <library name> <design files> ↵
```

If your design uses LPM, Altera megafunctions, or Altera primitives, you must also compile the Altera-provided functional models. The following commands show an example of each.

- Verilog HDL:

```
ncvlog -WORK lpm 220model.v ↵
ncvlog -WORK altera_mf altera_mf.v ↵
ncvlog -WORK altera altera_primitives.v ↵
```

If you are using the Quartus II software versions 5.0 and earlier and your design uses a memory initialization file, compile the **nopli.v** file, which is located in the <*Quartus II installation*>/**eda**/**sim_lib** directory, before you compile your model. For example:

```
ncvlog -WORK lpm nopli.v 220model.v ↵
ncvlog -WORK altera_mf nopli.v altera_mf.v ↵
```

Another option is to define `NO_PLI` during compilation with the following command:

```
ncvlog -DEFINE "NO_PLI=1" -WORK lpm 220model.v ↵
ncvlog -DEFINE "NO_PLI=1" -WORK altera_mf altera_mf.v ↵
```

- VHDL:

```
ncvhdl -V93 -WORK lpm 220pack.vhd ↵
ncvhdl -V93 -WORK lpm 220model.vhd ↵
ncvhdl -V93 -WORK altera_mf altera_mf_components.vhd ↵
ncvhdl -V93 -WORK altera_mf altera_mf.vhd ↵
ncvhdl -V93 -WORK altera altera_primitives_components.vhd ↵
```

```
ncvhdl -V93 -WORK altera altera_primitives.vhd ↵
```

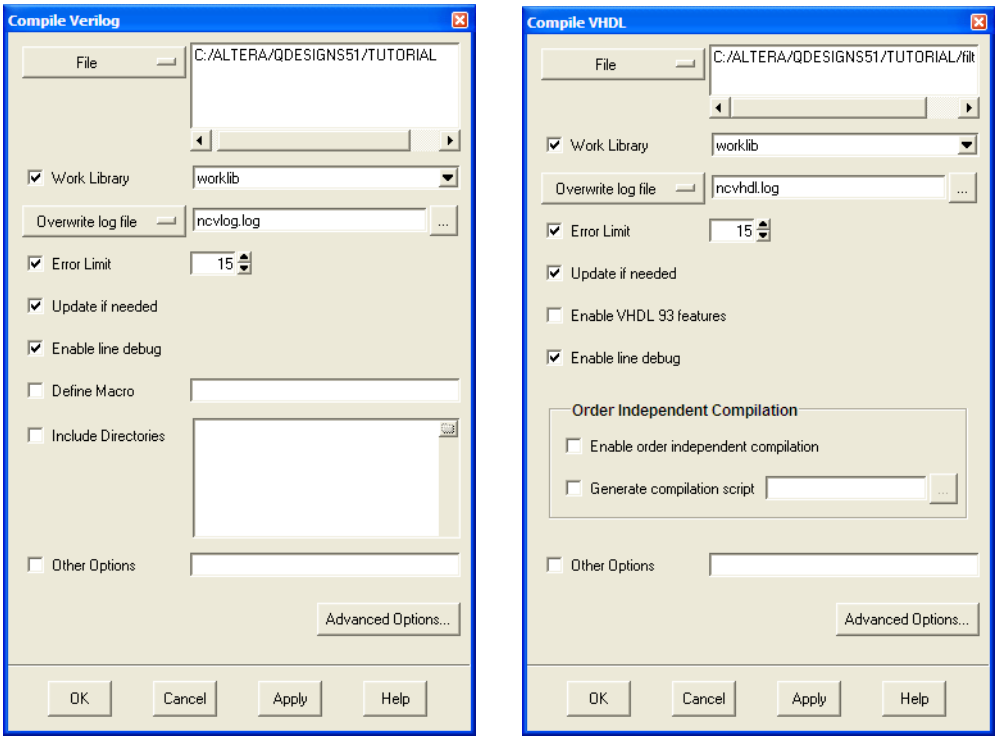
Compilation in GUI Mode

To compile using the NCLaunch GUI, perform the following steps:

1. Right-click a library filename in the NCLaunch window and click **NCVlog** (Verilog HDL) or **NCVhdl** (VHDL).

   Alternatively, on the Tools menu, click **Verilog Compiler** or **VHDL Compiler**. Figure 4–3 shows the **Compile Verilog** and **Compile VHDL** dialog boxes.

Figure 4–3. Compiling Verilog HDL and VHDL Files



2. Select the file and click **OK** in the **Compile Verilog** or **Compile VHDL** dialog box to begin compilation. The dialog box closes and returns you to NCLaunch.

The command-line equivalent argument is shown at the bottom of the NCLaunch window.

Elaborate Your Design

Before you can simulate your design, you must define the design hierarchy in a process called elaboration. When you use the Incisive simulator, you use the language-independent **ncelab** program to elaborate your design. The **ncelab** program constructs a design hierarchy based on the design's instantiation and configuration information, establishes signal connectivity, and computes initial values for all objects in the design. The elaborated design hierarchy is stored in a simulation snapshot, which is the representation of your design that the simulator uses to run the simulation. The snapshot is stored in the library database file, along with the other intermediate objects generated by the compiler and elaborator.

If you are running the NC-Verilog simulator with the single-step invocation method (**ncverilog**), and want to compile your source files and elaborate the design with one command, use the `+elaborate` option to stop the simulator after elaboration, for example:

```
ncverilog +elaborate test.v ↵
```

Elaboration in Command-Line Mode

To elaborate your Verilog HDL or VHDL design from the command line, use the following command:

```
ncelab [options] [<library>.]<cell>[:<view>] ↵
```

You can set your simulation timescale using the `-TIMESCALE` <*arguments*> option. The following example elaborates a dual-port RAM with the time scale option:

```
ncelab -TIMESCALE 1ps/1ps worklib.lpm_ram_dp_test:entity ↵
```

If you specified a timescale of 1 ps in the Verilog HDL testbench, the `TIMESCALE` option is not necessary. Using a ps resolution ensures the correct simulation of your design.

If your design includes high speed signals, you may need to add the following pulse reject options with your ncelab command.

```
ncelab -TIMESCALE 1ps/1ps worklib.mydesign:entity -PULSE_R 0 -PULSE_INT_R 0 ↵
```

For more information about the pulse reject options, refer to the *SDF Annotate Guide* from Cadence.

To list the elements in your library and the available views, use the **ncls** program. The following command displays all of the cells and their views in your current **worklib** directory:

```
ncls -library worklib ↵
```

For more information about the **ncls** program, refer to the Cadence NC-Verilog Simulator Help or Cadence NC-VHDL Simulator Help.

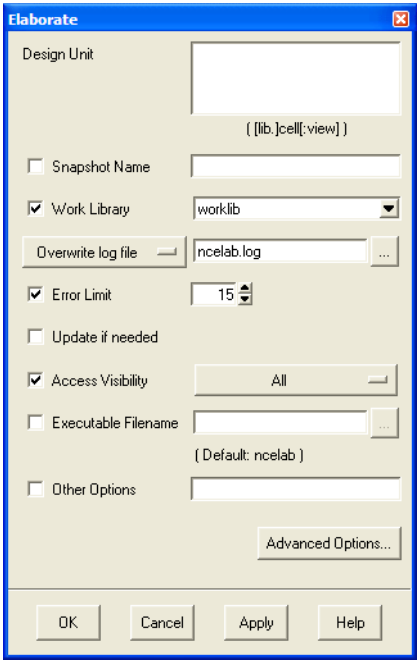
Elaboration in GUI Mode

To elaborate using the GUI, perform the following steps:

1. In the right side of the NCLaunch window, expand your current work library.
2. Select and expand (if necessary) the entity or module name you want to elaborate.
3. Right-click the view you want to display and click **NCElab**. The **Elaborate** dialog box appears (Figure 4–4). Optionally, on the Tools menu, click **Elaborator**.
4. In the **Other Options** box, set the simulation timescale by typing (Figure 4–4):

   ```
   -TIMESCALE 1ps/1ps
   ```

Figure 4–4. Elaborating the Design



5. Click **OK** in the **Elaborate** dialog box to begin elaboration. The dialog box closes and returns you to NCLaunch.

Add Signals to View

To view the stored selected signals, use an SHM database, which is a Cadence proprietary waveform database, to store the selected signals you want to view. Before you can specify which signals to view, you must create the database by adding commands to your code. Or you can create a Value Change Dump File (**.vcd**) to store the simulation history.

For more information about using a Value Change Dump File, refer to the *Cadence NC-Sim User Manual* from Cadence included with the installation.

Adding Signals in Command-Line Mode

To create an SHM database, specify the system tasks described in Table 4–5 in your Verilog HDL code.

For VHDL, you can use the Tcl command interface or C function calls to add signals to a database. Refer to the Cadence documentation included in the installation package for details.

Table 4–5. SHM Database System Tasks

System Task	Description
`$shm_open("<filename>.shm");`	Opens a database. If you do not specify a filename, the default **waves.shm** database opens. If a database with the specified name does not exist, it is created for you.
`$shm_probe("[A|S|C]");`	Probe signals. You can specify the signals to probe; if you do not specify signals, the default is all ports in the current scope. `A` probes all nodes in the current scope. `S` probes all nodes below the current scope. `C` probes all nodes below the current scope and in libraries.
`$shm_save;`	Saves the database.
`$shm_close;`	Closes the database.

The following sample shows a simple example of how to add signals to an SHM database.

```
initial
    begin
        $shm_open ("waves.shm");
        $shm_probe ("AS");
    end
```

You can insert this code sample into your Verilog HDL file. It is applicable only for Verilog HDL files. For more information about these system tasks, refer to the Cadence NC-Sim software user manual included in the installation.

Adding Signals in GUI Mode

To add signals in GUI mode, perform the following steps:

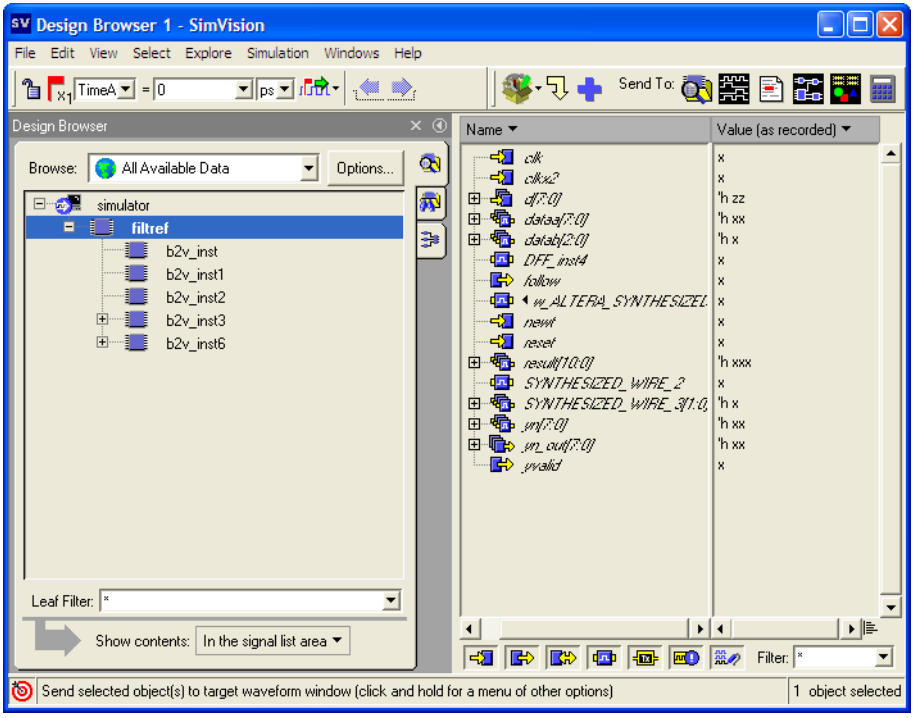
1. In the NC-Sim software, load the design.
    a. In the NCLaunch window, click the **+** icon to expand the **Snapshots** directory.
    b. Right-click on the **lib.cell:view** you want to simulate and click **NCSim**.

c. Click **OK** in the **Simulate** dialog box.

After you load the design, the SimVision Console and SimVision Design Browser windows appear. Figure 4–5 shows the SimVision Design Browser window.

2. In the Design Browser window, select a module in the left side of the window to display the signal names (Figure 4–5).

Figure 4–5. SimVision Design Browser



3. To send the selected signals to the Waveform Viewer, perform one of the following steps:

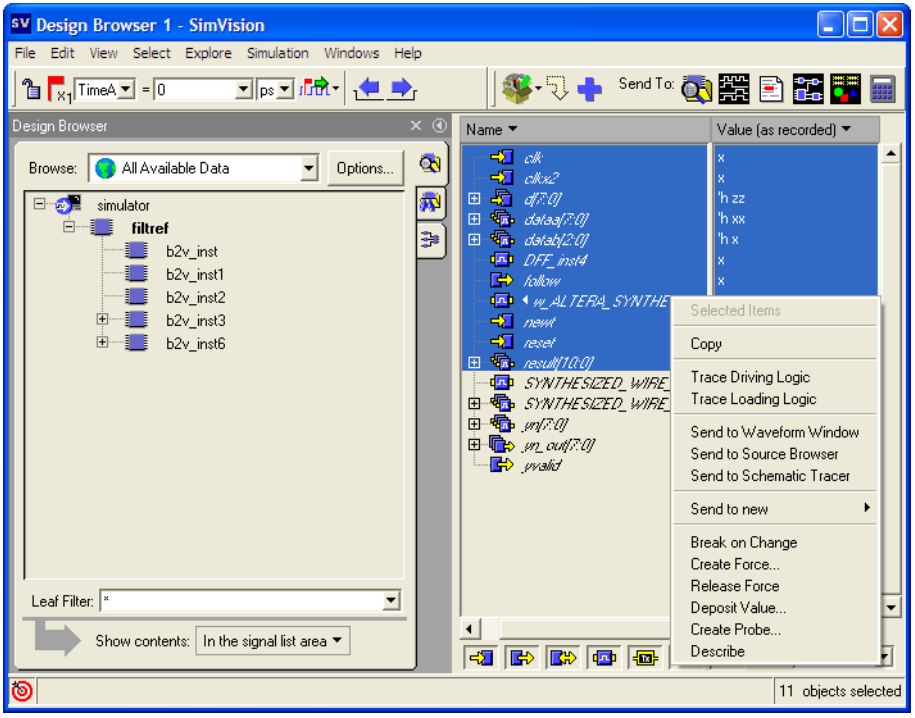
Select a group of signals from the right side of the Design Browser window and click the **Send to Waveform Viewer** icon in the **Send To** toolbar (the upper-right area of the Design Browser window).

or

Right-click the signals and click **Send to Waveform Window** (Figure 4–6).

A waveform window showing all of your signals appears. You are now ready to simulate your testbench and design.

Figure 4–6. Selecting Signals in the Design Browser Window



Simulate Your Design

After you have compiled and elaborated your design, you can simulate it using **ncsim**. The **ncsim** program loads the file or snapshot generated by **ncelab** as its primary input. It then loads other intermediate objects referenced by the snapshot. If you enable interactive debugging, it may also load HDL source files and script files. The simulation output is controlled by the model or debugger. The output can include result files generated by the model, SHM database, or Value Change Dump File.

Functional/RTL Simulation in Command-Line Mode

To perform functional/RTL simulation of your Verilog HDL or VHDL design at the command line, type the following command:

```
ncsim [options] [<library>.]<cell>[:<view>] ↵
```

For example:

```
ncsim worklib.lpm_ram_dp:syn ↵
```

Table 4–6 shows some of the options you can use with **ncsim**.

Table 4–6. ncsim Options

Options	Description
`-gui`	Launch GUI mode
`-batch`	Used for non-interactive mode
`-tcl`	Used for interactive mode (not required when using `-gui`)

Functional/RTL Simulation in GUI Mode

You can run and step through simulation of your Verilog HDL or VHDL design in the GUI. In the Design Browser window, on the Simulation menu, click **Run** to begin the simulation.



You must load the design before simulating. If you have not done so, refer to step 1 in "Adding Signals in GUI Mode" on page 4–18 for instructions.

Post-Synthesis Simulation

The following sections provide detailed instructions for performing post-synthesis simulation using Quartus II output files, simulation libraries, and the Incisive platform software.

Quartus II Simulation Output Files

After performing synthesis with either a third-party synthesis tool or with the Quartus II integrated synthesis, you must generate a simulation netlist for functional simulations. To generate a simulation netlist for functional simulation, perform the following steps in the Quartus II software:

1. Perform Analysis and Synthesis. On the Processing menu, point to Start and click **Start Analysis and Synthesis**.
2. Turn on the **Generate Netlist for Functional Simulation Only** option by performing the following steps:
    a. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
    b. In the **Category** list, select **Simulation**. The **Simulation** page appears.
    c. In the **Tool name** list, select **NCSim**.
    d. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
    e. Click **More Settings**. The **More EDA Tools Simulation Settings** dialog box appears. In the **Existing options settings list**, click **Generate Netlist for Functional Simulation Only** and select **On** from the **Setting** list under **Option**.
    f. Click **OK**.
    g. In the **Settings** dialog box, click **OK**.
3. Run the EDA Netlist Writer. On the Processing menu, point to Start and click **Start EDA Netlist Writer**.



During the EDA Netlist Writer stage, the Quartus II software produces a Verilog Output File (**.vo**) or VHDL Output File (**.vho**) that can be used for post-synthesis simulations in the NC-Sim software. This netlist file is mapped to architecture-specific primitives. No timing information is included at this stage. The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to the *<project directory>***/simulation/NCSim** directory.

Create Libraries

Create the following libraries for your simulation:

- Work library
- Device family library targeting your design targets using the following files in the *<path to Quartus II installation>*/**eda**/**sim_lib** directory:
  - *<device_family>***_atoms.v**
  - *<device_family>***_atoms.vhd**
  - *<device_family>***_components.vhd**

Compile Project Files and Libraries

Compile the project files and libraries into your work directory using the **ncvlog** or **ncvhdl** programs or the GUI. Include the following files:

- Test bench file
- The Quartus II software functional output netlist file (Verilog Output File or VHDL Output File)
- Atom library file for the device family *<device family>***_atoms.***<v|vhd>*
- For VHDL, *<device family>***_components.vhd**

Refer to the section "Compile Source Code and Testbenches" on page 4–13 for instructions about compiling.

Elaborate Your Design

Elaborate your design using the **ncelab** program as described in "Elaboration in GUI Mode" on page 4–16.

Add Signals to the View

Refer to the section "Add Signals to View" on page 4–17 for information about adding signals to the view.

Simulate Your Design

Simulate your design using the **ncsim** program as described in "Simulate Your Design" on page 4–20.

Gate-Level Timing Simulation

The following sections provide detailed instructions for performing timing simulation using the Quartus II output files, simulation libraries, and Cadence NC tools.

Generating a Gate-Level Timing Simulation Netlist

To perform gate-level timing simulation, the NC-Sim software requires information about how the design was placed into device-specific architectural blocks. The Quartus II software provides this information in the form of a Verilog Output File for Verilog HDL designs and a VHDL Output File for VHDL designs. The accompanying timing information is stored in the SDO file, which annotates the delay for the elements found in the Verilog Output File or VHDL Output File.

To generate the Verilog Output File or VHDL Output Files and the Standard Delay File, perform the following steps:

1. Perform a full compilation. On the Processing menu, click **Start Compilation**.
2. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
3. In the **Category** list, select **Simulation**. The **Simulation** page appears.
4. In the **Tool name** list, select **NCSim**.
5. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
6. Click **OK**.
7. Run the EDA Netlist Writer. On the Processing menu, point to Start and click **Start EDA Netlist Writer**.

   During the EDA Netlist Writer stage, the Quartus II software produces a Verilog Output File (**.vo**), VHDL Output File (**.vho**), and a SDO file used for gate-level timing simulations in the NC-Sim software. This netlist file is mapped to architecture-specific

primitives. The timing information for the netlist is included in the SDO file. The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to the *<project directory>***/simulation/ncsim** directory.

Generating a Different Timing Model

If you enable the Quartus II Classic Timing Analyzer or Quartus II TimeQuest Timing Analyzer when generating the SDO file, slow-corner (worst case) timing models are used by default. To generate the SDO file using a different timing model, you must run the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer with a different timing model before you start the EDA Netlist Writer.

To run the Quartus II Classic Timing Analyzer with the best-case model, on the Processing menu, point to Start and click **Start Classic Timing Analyzer (Fast Timing Model)**. After timing analysis is complete, the Compilation Report appears. You can also type the following at a command prompt:

```
quartus_tan <project_name> --fast_model=on ↵
```

To run the Quartus II TimeQuest Timing Analyzer with a best-case model, use the `-fast_model` option after you create the timing netlist. The following command enables the fast timing models:

```
create_timing_netlist -fast_model
```

You can also type the following command at a command prompt:

```
quartus_sta <project_name> --fast_model=on ↵
```

For more information about generating the timing model, refer to the *Quartus II Classic Timing Analyzer* or *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

After you run the Quartus II Classic Timing Analyzer or Quartus II TimeQuest Timing Analyzer, you can perform steps 2 through 7 in "Generating a Gate-Level Timing Simulation Netlist" on page 4–24 to generate the SDO file. For fast corner timing models, the _fast post fix is added to the VO, VHO, and SDO file (for example, **my_project_fast.vo**, **my_project_fast.vho**, and **my_project_fast.sdo**).

Operating Condition Example: Generate All Timing Models for Stratix III and Cyclone III Devices

In Stratix® III and Cyclone® III devices, you can specify different temperature and voltage parameters to generate the timing models. Table 4–7 shows the available operating conditions (model, voltage, and temperature) for Stratix III and Cyclone III devices.

Table 4–7. Available Operating Conditions for Stratix III and Cyclone III Devices

Device Family	Model	Voltage	Temperature
Stratix III	Slow	1100 mV	85° C
	Slow	1100 mV	0° C
	Fast	1100 mV	0° C
Cyclone III	Slow	1200 mV	85° C
	Slow	1200 mV	0° C
	Fast	1200 mV	0° C

To generate the SDO files for the three different operating conditions for a Stratix III design, perform the following steps:

1. Generate all the available corner models at all operating conditions. Type the following command at a command prompt:

```
quartus_sta <project name> --multicorner ↵
```

2. Generate the ModelSim simulation output files for all three corners specified above. The output files are generated in the simulation output directory. Type the following command at a command prompt:

```
quartus_eda <project name> --simulation --tool=ncsim --format=verilog
```

To summarize, for the three operating conditions the steps above generate the following files in the simulation output directory:

First slow corner (slow, 1100 mV, 85° C):
VO file— <*revision name*>**.vo**
SDO file— <*revision name*>**_v.sdo**

Second slow corner (slow, 1100 mV, 0º C):
VO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_slow.vo**
SDO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_v_slow.sdo**

Fast corner (fast, 1100 mV, 0º C):
VO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_fast.vo**
SDO file— <*revision name*>_<*speedgrade*>**_1100mv_0c_v_fast.sdo**

Perform Timing Simulation Using Post-Synthesis Netlist

You can perform a timing simulation using the post-synthesis netlist instead of using a gate-level netlist and you can generate a Standard Delay Format Output File (**.sdo**) without running the Fitter. In this case, the SDO file includes all timing values for the device cells only. Interconnect delays are not included because fitting (placement and routing) has not been performed.

To generate the post-synthesis netlist and the SDO file, type the following at a command prompt:

```
quartus_map <project name> -c <revision name> ↵
quartus_tan <project name> -c <revision name> --post_map --zero_ic_delays ↵
quartus_eda <project name> -c <revision name> --simulation --tool= \
   <3rd party EDA tool> --format=<HDL language> ↵
```

For more information on the `-format` and `-tool` options, type the following command at a command prompt:

```
quartus_eda -help=<options> command ↵
```

Quartus II Timing Simulation Libraries

Altera device simulation library files are provided in the <*Quartus II installation*>/**eda**/**sim_lib** directory. The Verilog Output File or VHDL Output File requires the library for the device your design targets. For example, the Stratix device family requires the following library files:

- **stratix_atoms.v**
- **stratix_atoms.vhd**
- **stratix_components.vhd**

If your design targets a Stratix device, you must set up the appropriate mappings in your **cds.lib** file. Refer to "Create Libraries" for more information.

Create Libraries

Create the following libraries for your simulation:

- Work library
- Device family libraries targeting using the following files in the <*path to Quartus II installation*>/**eda**/**sim_lib** directory:
  - <*device_family*>**_atoms.v**
  - <*device_family*>**_atoms.vhd**
  - <*device_family*>**_components.vhd**

For step-by-step instructions on creating libraries, refer to "Basic Library Setup" on page 4–7 and "LPM Functions, Altera Megafunctions, and Altera Primitives Libraries" on page 4–9.

Compile the Project Files and Libraries

Compile the project files and libraries into your work directory using the **ncvlog** or **ncvhdl** programs or the GUI. Include the following files:

- Test bench file
- The Quartus II software functional output netlist file (Verilog Output File or VHDL Output File)
- Atom library file for the device family <*device family*>**_atoms.**<***v*** | ***vhd***>
- For VHDL, <*device family*>**_components.vhd**

For instructions on compiling, refer to "Compile Source Code and Testbenches" on page 4–13.

Elaborate Your Design

When performing elaboration with the Quartus II-generated Verilog HDL netlist file, the Standard Delay Format Output File is read automatically. When you run **ncelab**, it recognizes the embedded system task $sdf_annotate and automatically compiles and annotates the Standard Delay Format Output File (runs **ncsdfc** automatically).



The Standard Delay Format Output File should be located in the same directory where you invoke an elaboration or simulation, because the `$sdf_annotate` task references the Standard Delay Format Output File without using a full path. If you are invoking an elaboration or simulation from a different directory, you can either comment out the `$sdf_annotate` and annotate the Standard Delay Format Output File with the GUI, or add the full path of the Standard Delay Format Output File.

Refer to "Elaborate Your Design" on page 4–15 for step-by-step instructions on elaboration.

For VHDL, the Quartus II software-generated VHDL netlist file does not contain system task calls to locate your SDF file; therefore, you must compile the Standard Delay Format Output File manually. Refer to "Compiling the Standard Delay Output File (VHDL Only) in Command-Line Mode" and "Compiling the Standard Delay Output File (VHDL Only) in GUI Mode" for information about compiling the Standard Delay Format Output File.

Compiling the Standard Delay Output File (VHDL Only) in Command-Line Mode

To annotate the Standard Delay Format Output File timing data from the command line, perform the following steps:

1. Compile the Standard Delay Format Output File using the **ncsdfc** program by typing the following command at the command prompt:

   `ncsdfc <project name>_vhd.sdo -output <output name>` ↵

   The **ncsdfc** program generates an <*output name*>**.sdf.X** compiled SDO.

   If you do not specify an output name, **ncsdfc** uses <*project name*>**.sdo.X**.

2. Specify the compiled Standard Delay Format Output File for the project by adding the following command to an ASCII SDF command file for the project:

```
COMPILED_SDF_FILE = "<project name>.sdf.X" SCOPE = <instance path>
```

The following code shows an example of an SDF command file:

```
// SDF command file sdf_file
COMPILED_SDF_FILE = "lpm_ram_dp_test_vhd.sdo.X",
SCOPE = :tb,
MTM_CONTROL = "TYPICAL",
SCALE_FACTORS = "1.0:1.0:1.0",
SCALE_TYPE = "FROM_MTM";
```

After you compile the Standard Delay Format Output File, run the following command to elaborate the design:

`ncelab worklib.<project name>:entity -SDF_CMD_FILE <SDF Command File>` ↵

Compiling the Standard Delay Output File (VHDL Only) in GUI Mode

To annotate the SDO file timing data in the GUI, perform the following steps in the NCLaunch window:

1. On the Tools menu, click **SDF Compiler**. The **Compile SDF** dialog box appears.
2. In the **SDF File** box, type in the name of the Standard Delay Format Output File (**.sdo**) for the project.
3. Click **OK**.

   When the Standard Delay Format Output File compilation is complete, you can elaborate the design. Refer to "Elaboration in GUI Mode" on page 4–16 for step-by-step instructions.

   If you are performing a VHDL gate-level simulation, you must create an SDF command file before you begin elaboration. To create the SDF command file, perform steps 5 through 11.

4. On the Tools menu, click **Elaborator**. The **Elaborate** dialog box appears.
5. Click **Advanced Options**.
6. Click **Annotation**.
7. Turn on **Use SDF File**.
8. Click **Edit**.
9. Browse to the location of the SDF command file name.
10. Click **Add** and browse to the location of the Standard Delay Format Output File in the **Compiled SDF File** box and click **OK**.
11. Click **OK** to save and exit the **SDF Command File** dialog box.

Add Signals to View

Refer to the section "Add Signals to View" on page 4–17 for information about adding signals to view.

Simulate Your Design

Simulate your design using the **ncsim** program as described in "Simulate Your Design" on page 4–20.



For the design examples to run gate-level timing simulation, refer to **www.altera.com/support/examples/ncsim/exm-ncsim.html**.

Simulating Designs that Include Transceivers

If your design includes a Stratix II GX or Stratix GX transceiver, you must compile additional library files to perform functional or timing simulations.

Stratix GX Functional Simulation

To perform a functional simulation of your design that instantiates the `altgxb` megafunction, enabling the gigabit transceiver block (GXB) on Stratix GX devices, compile the **stratixgx_mf** model file into the **altgxb** library.



The **stratixgx_mf** model file references the **lpm** and **sgate** libraries, so you will need to create these libraries to perform a simulation.

Example of Compiling Library Files for Functional Stratix GX Simulation in Verilog HDL

To compile the libraries necessary for a functional simulation of a Verilog HDL design targeting a Stratix GX device, type the following commands at the NC Sim command prompt:

```
ncvlog -work lpm 220model.v ↵
ncvlog -work altera_mf altera_mf.v ↵
ncvlog -work sgate sgate.v ↵
ncvlog -work altgxb stratixgx_mf.v ↵
ncsim work.<my design> ↵
```

Example of Compiling Library Files for Functional Stratix GX Simulation in VHDL

To compile the libraries necessary for functional simulation of a VHDL design targeting a Stratix GX device, type the following commands at the NC-Sim command prompt:

```
ncvhdl -work lpm 220pack.vhd 220model.vhd ↵
ncvhdl -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
ncvhdl -work sgate sgate_pack.vhd sgate.vhd ↵
ncvhdl -work altgxb stratixgx_mf.vhd stratixgx_mf_components.vhd ↵
```

```
ncsim work.<my design> ↵
```

Stratix GX Post-Fit (Timing) Simulation

To perform a post-fit timing simulation of your design that includes a Stratix GX transceiver, compile the **stratixgx_atoms** and **stratixgx_hssi_atoms** model files into the **stratixgx** and **stratixgx_gxb** libraries, respectively.



You need to create these libraries to perform a simulation because the **stratixgx_hssi_atoms** model file references the **lpm** and **sgate** libraries.

Example of Compiling Library Files for Timing Stratix GX Simulation in Verilog HDL

To compile the libraries necessary to timing simulation of a Verilog HDL design targeting a Stratix GX device, type the following commands at the NC-Sim command prompt:

```
ncvlog -work lpm 220model.v ↵
ncvlog -work altera_mf altera_mf.v ↵
ncvlog -work sgate sgate.v ↵
ncvlog -work stratixgx stratixgx_atoms.v ↵
ncvlog -work stratixgx_gxb stratixgx_hssi_atoms.v ↵
ncelab work.<my design> -TIMESCALE 1ps/1ps -PULSE_R 0 \
-PULSE_INT_R 0 ↵
```

Example of Compiling Library Files for Timing Stratix GX Simulation in VHDL

To compile the libraries necessary for timing simulation of a VHDL design targeting a Stratix GX device, type the following commands at the NC-Sim command prompt:

```
ncvhdl -work lpm 220pack.vhd 220model.vhd ↵
ncvhdl -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
ncvhdl -work sgate sgate_pack.vhd sgate.vhd ↵
ncvhdl -work stratixgx stratixgx_atoms.vhd stratixgx_components.vhd ↵
ncvhdl -work stratixgx_gxb stratixgx_hssi_atoms.vhd \
stratixgx_hssi_components.vhd ↵
ncelab work.<my design> -TIMESCALE 1ps/1ps -PULSE_R 0 -PULSE_INT_R 0 ↵
```

Stratix II GX Functional Simulation

To perform a post-fit timing simulation of your design that instantiates the `alt2gxb` megafunction, edit your **cds.lib** file so that all the libraries point to the work library, and compile the **stratixiigx_hssi** model file into the **stratixiigx_hssi** library. When compiling the library files, you can safely ignore the following warning message:

```
"Multiple logical libraries mapped to a single location"
```

The following example is of the **cds.lib** file.

Example 4–1. Example of a cds.lib File

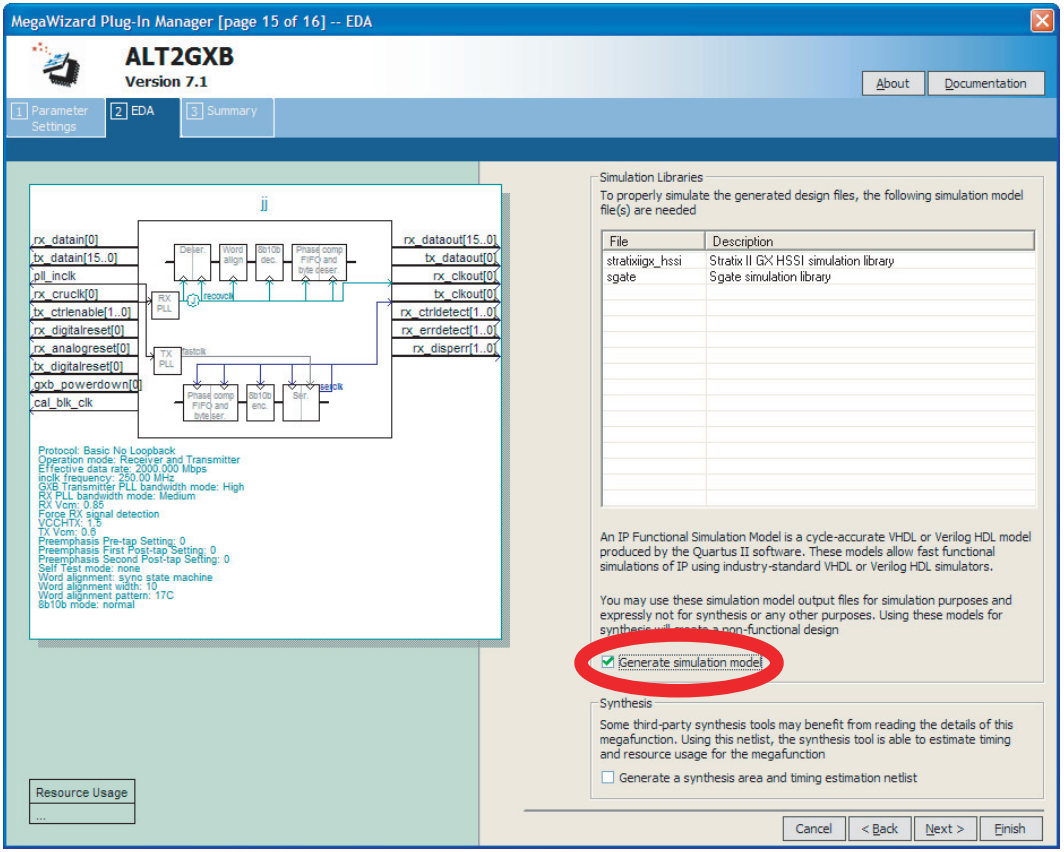
```
SOFTINCLUDE ${CDS_INST_DIR}/tools/inca/files/cdsvhdl.lib
SOFTINCLUDE ${CDS_INST_DIR}/tools/inca/files/cdsvlog.lib
DEFINE work ./ncsim_work
DEFINE stratixiigx_hssi ./ncsim_work
DEFINE stratixiigx ./ncsim_work
DEFINE lpm ./ncsim_work
DEFINE sgate ./ncsim_work
```



The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries, so you will need to create these libraries to perform a simulation.

Generate a functional simulation netlist by turning on **Create a simulation library for this design** in the last page of the alt2gxb MegaWizard (Figure 4–7). The *<alt2gxb entity name>*.**vho** or *<alt2gxb module name>*.**vo** is generated in the current project directory.

Figure 4–7. alt2gxb MegaWizard



The Quartus II generated alt2gxb functional simulation library file references stratixiigx_hssi wysiwyg atoms.

Example of Compiling Library Files for Functional Stratix II GX Simulation in Verilog HDL

To compile the libraries necessary to functional simulation of a Verilog HDL design targeting a Stratix II GX device, type the following commands at the NC-Sim command prompt:

```
ncvlog -work lpm 220model.v ↵
ncvlog -work altera_mf altera_mf.v ↵
ncvlog -work sgate sgate.v ↵
ncvlog -work stratixiigx_hssi stratixiigx_hssi_atoms.v ↵
```

```
ncvlog -work work <alt2gxb module name>.vo ↵
ncelab work.<my design> ↵
```

Example of Compiling Library Files for Functional Stratix II GX Simulation in VHDL

To compile the libraries necessary for functional simulation of a VHDL design targeting a Stratix II GX device, type the following commands at the NC-Sim command prompt:

```
ncvhdl -work lpm 220pack.vhd 220model.vhd ↵
ncvhdl -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
ncvhdl -work sgate sgate_pack.vhd sgate.vhd ↵
ncvhdl -work stratixiigx_hssi stratixiigx_hssi_components.vhd \
stratixiigx_hssi_atoms.vhd ↵
ncvhdl -work work <alt2gxb entity name>.vho ↵
ncelab work.<my design> ↵
```

Stratix II GX Post-Fit (Timing) Simulation

To perform a post-fit timing simulation of your design that includes the `alt2gxb` megafunction, edit your **cds.lib** file so that all the libraries point to the work library and compile `stratixiigx_atoms` and `stratixiigx_hssi_atoms` into the **stratixiigx** and **stratixiigx_hssi** libraries, respectively. When compiling the library files, you can safely ignore the following warning message:

```
"Multiple logical libraries mapped to a single location"
```

For an example of a **cds.lib** file, refer to "Stratix II GX Functional Simulation" on page 4–33.



The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries, so you will need to create these libraries to perform a simulation.

Example of Compiling Library Files for Timing Stratix II GX Simulation in Verilog HDL

To compile the libraries necessary to timing simulation of a Verilog HDL design targeting a Stratix II GX device, type the following commands at the NC-Sim command prompt:

```
ncvlog -work lpm 220model.v ↵
ncvlog -work altera_mf altera_mf.v ↵
ncvlog -work sgate sgate.v ↵
ncvlog -work stratixiigx  stratixiigx_atoms.v ↵
ncvlog -work stratixiigx_hssi stratixiigx_hssi_atoms.v ↵
```

```
ncelab work.<my design> -TIMESCALE 1ps/1ps -PULSE_R 0 -PULSE_INT_R 0 ↵
```

Example of Compiling Library Files for Timing Stratix II GX Simulation in VHDL

To compile the libraries necessary for timing simulation of a VHDL design targeting a Stratix II GX device, type the following commands at the NC-Sim command prompt:

```
ncvhdl -work lpm 220pack.vhd 220model.vhd ↵
ncvhdl -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
ncvhdl -work sgate sgate_pack.vhd sgate.vhd ↵
ncvhdl -work stratixiigx stratixiigx_atoms.vhd \
stratixiigx_components.vhd ↵
ncvhdl -work stratixiigx_hssi stratixiigx_hssi_components.vhd \
stratixiigx_hssi_atoms.vhd ↵
ncvhdl -work work <alt2gxb>.vho ↵
ncelab work.<my design> -TIMESCALE 1ps/1ps -PULSE_R 0 -PULSE_INT_R 0 ↵
```

Pulse Reject Delays

By default, the NCSim software filters out all pulses that are shorter than the propagation delay between primitives. Setting the pulse reject delays (similar to transport delays) options in the NC-Sim software prevents the simulation tool from filtering out these pulses. Use the following options to ensure that all signal pulses are seen in the simulation results.

-PULSE_R

Use this option when the pulses in your simulation are shorter than the delay within a gate-level primitive. The argument is the percentage of delay for pulse reject limit for the path.

-PULSE_INT_R

Use this option when the pulses in your simulation are shorter than the interconnect delay between gate-level primitives. The argument is the percentage of delay for pulse reject limit for the path. The `-PULSE_R` and `-PULSE_INT_R` options are also used by default in the NativeLink® feature for gate-level timing simulation.

The following NC-Sim software command describes the command-line syntax to perform a gate-level timing simulation with the device family library:

```
ncelab worklib.<project name>:entity -SDF_CMD_FILE <SDF Command File> \
-TIMESCALE 1ps/1ps -PULSE_R 0 -PULSE_INT_R 0
```

Using the NativeLink Feature with NC-Sim

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools and allows you to run NC-Sim within the Quartus II software.

Setting Up NativeLink

To run NC-Sim automatically from the Quartus II software using the NativeLink feature, you must specify the path to your simulation tool by performing the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, select **EDA Tool Options**. The **EDA Tool Options** page is shown.
3. Double-click the entry under the **Location of executable** column beside the name of your **EDA Tool**, and type or browse to the directory containing the executables of your EDA tool.
4. Click **OK**.

You can also specify the path to the simulator's executables by using the `set_user_option` TCL command:

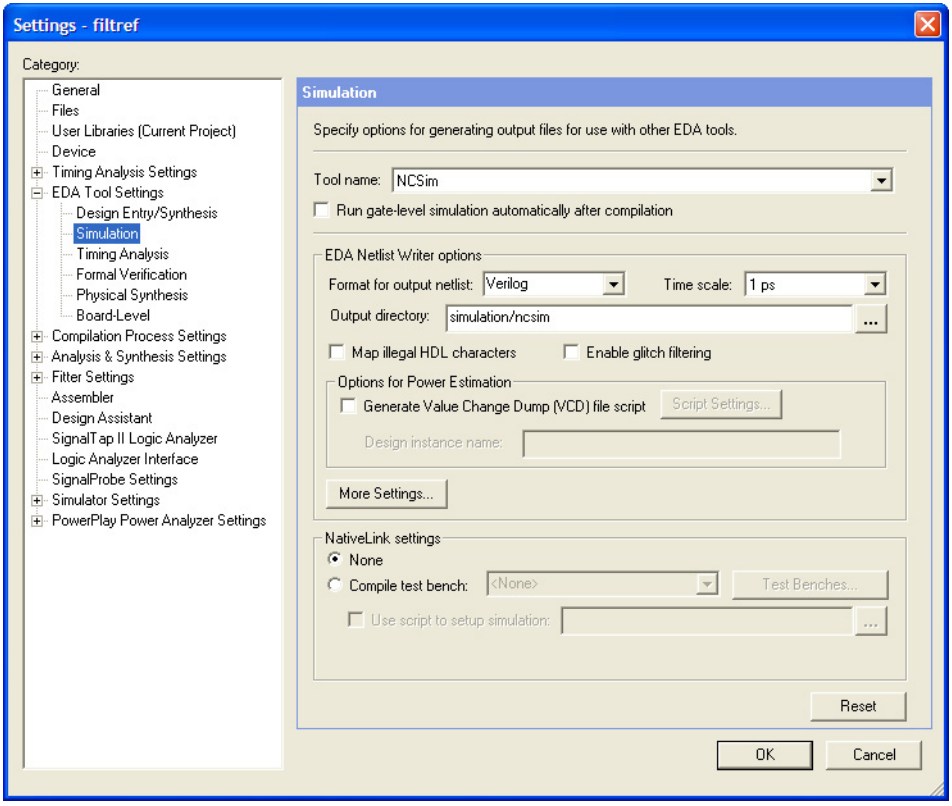
```
set_user_option -name EDA_TOOL_PATH_NCSIM <path to executables>
```

Performing an RTL Simulation Using NativeLink

To run a functional RTL simulation with the NC-Sim software automatically in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears (Figure 4–8).

Figure 4–8. Simulation Page in the Settings Dialog Box



3. In the **Tool name** list, select **NCSim**.

4. If your design is written entirely in Verilog HDL or in VHDL, the NativeLink feature automatically chooses the correct language and Altera simulation libraries. If your design is written with mixed languages, the NativeLink feature uses the default language specified in the **Format for output netlist** list. To change the default

language when there is a mixed language design, under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. Table 4–8 shows the design languages for output netlists and simulation models.

Table 4–8. NativeLink Design Languages

Design File	Format for Output Netlist	Simulation Models Used
Verilog	Any	Verilog
VHDL	Any	VHDL
Mixed	Verilog	Verilog
Mixed	VHDL	VHDL



For mixed language simulation, choose the same language that was used to generate your megafunctions to ensure correct parameter passing between the megafunctions and the Altera libraries. For example, if your altsyncram megafunction was generated in VHDL, choose VHDL as the format for output netlist.

For mixed language simulations, it is important to be aware of the following conditions:

- VHDL designs instantiating Verilog user-defined primitives (UDPs) are not supported.
- Parameters cannot be passed in Verilog modules that instantiate VHDL components.

5. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.

   For more information about setting up a testbench with NativeLink, refer to the section "Setting Up a Testbench" on page 4–40.

6. Click **OK.**

7. On the Processing menu, point to Start and click **Start Analysis and Elaboration** to perform an analysis and elaboration. This command collects all your file name information and builds your design hierarchy in preparation for simulation.

8. On the Tools menu, point to **EDA Simulation Tool** and click **Run EDA RTL Simulation** to automatically run NC-Sim, compile all necessary design files, and complete a simulation.

Performing a Gate Level Simulation Using NativeLink

To run a gate-level timing simulation with the NC-Sim software in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** appears (Figure 4–8 on page 4–38).
3. In the **Tool name** list, select **NCSim**.
4. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, choose **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
5. To perform a gate level simulation after each full compilation, turn on **Run Gate Level Simulation automatically after compilation**.
6. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.
7. Click **OK**.
8. On the Processing menu, point to Start and click **Start EDA Netlist Writer** to generate a simulation netlist of your design.
9. On the Tools menu, point to EDA Simulation Tool and click **Run EDA Gate Level Simulation** to automatically run NC-Sim, compile all necessary design files, and complete a simulation.



A Tcl File (*.tcl) is generated in the *<project_directory>***simulation\ncsim** directory when you run NativeLink. This TCL File enables you to simulate the design with the following command without using NativeLink:

```
quartus_sh -t <project_directory>\simulation\ncsim\<generated_do_file>.tcl
```

Setting Up a Testbench

You can compile your design files and testbench files, and run EDA simulation tools to perform a simulation automatically using the NativeLink feature.

To setup NativeLink with a testbench, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears.
3. Under **NativeLink settings**, select **None** or **Compile test bench** (Table 4–9).

Table 4–9. NativeLink Settings

Settings	Description
None	Compile simulation models and design files.
Compile test bench	NativeLink compiles simulation models, design files, testbench files, and starts simulation.

4. If you select **Compile test bench**, select your testbench setup from the **Compile test bench** list. You can use different testbench setups to specify different testbench files for different test scenarios. If there are no testbench setups entered, create a testbench setup by performing the following steps:
   a. Click **Test Benches**. The **Test Benches** dialog box appears.
   b. Click **New**. The **New Test Bench Settings** dialog box appears.
   c. In the **Test Bench name** box, type in the testbench setup name which is used to identify the different testbench setups.
   d. In the **Test bench entity** box, type in the top-level entity name. For example, for a Quartus II generated VHDL testbench, type `filtref_vhd_vec_tst`.
   e. In the **Instance** box, type in the full instance path to the top level of your FPGA design. For example, for a Quartus II generated VHDL testbench, type `i1`.
   f. Under **Simulation period**, select **Run simulation until all vector stimuli are used**. If you select **End simulation at**, specify the simulation end time and the time unit.

g. Under **Test bench files**, browse and add all your testbench files in the **File name** box. Use the **Up** and **Down** button to reorder your files. The script used by NativeLink compiles the files in the order from top to the bottom.



You can also specify the library name and the HDL version to compile the testbench file. NativeLink compiles the testbench to the library name of the HDL specified version.

h. Click **OK**.

i. In the **Test Benches** dialog box, click **OK**.

5. Under **NativeLink settings**, you can turn on **Use script to setup simulation** and browse to your script. You can write a script to setup your waveforms before running the simulation.



The script should be a valid NC-Sim tcl script. NativeLink passes the script to ncsim command with command-line arguments to set up and run simulation.

Creating a Testbench

In the Quartus II software, you can create a Verilog HDL or VHDL testbench from a Vector Waveform File. The generated testbench includes the behavior of the input stimulus and applies it to your instantiated top-level FPGA design.

1. On the File menu, click **Open**. The **Open** dialog box appears.
2. Click the **Files of type** arrow and select **Waveform/Vector Files**. Select your file.
3. Click **Open**.
4. On the File menu, click **Export**. The **Export** dialog box.
5. Click the **Save as type** arrow and select **VHDL Test Bench File (*.vht)** or **Verilog Test Bench File (*.vt)**.
6. You can turn on **Add self-checking code to file** to check your simulation results against your Vector Waveform File.
7. Click **Export**.

Incorporating PLI Routines

Designers frequently use PLI routines in Verilog HDL testbenches to perform user- or design-specific functions that are beyond the scope of the Verilog HDL language. Cadence NC simulators include the PLI wizard, which helps you incorporate your PLI routines.

For example, if you are using the Quartus II software version 5.0 and earlier, and you are using a Hexadecimal (Intel-Format) File for memory, you can convert it for use with NC tools using the Altera-provided **convert_hex2ver** function. To use this function, you must build it and place it in your project directory using the PLI wizard.

This section describes how to dynamically link, dynamically load, and statically link a PLI library using the **convert_hex2ver** function as an example. The following **convert_hex2ver** source files are located in the *<path to Quartus II installation>*/**eda**/**cadence**/**verilog-xl** directory:

- **convert_hex2ver.c**
- **veriuser.c**

Dynamically Link a PLI Library

To create a PLI dynamic library (**.so** or **.sl**), perform the following steps:

1. Run the PLI wizard by typing `pliwiz` at the command prompt.
2. In the **Config Session Name and Directory** page, type the name of the session in the **Config Session Name** box and type the directory in which the file should be built in the **Config Session Directory** box.
3. Click **Next**.
4. In the **Select Simulator/Dynamic Libraries** page, turn on the **Dynamic Libraries Only** option.
5. Click **Next**.
6. In the **Select Components** page, select the **PLI 1.0 Applications** option, and then select **libpli**.
7. Click **Next**.
8. In the **Select PLI 1.0 Application Input** page, select **Existing VERIUSER** (source/object file).
9. Select **Source File** and click **Browse** to locate the **veriuser.c** file provided with the Quartus II software.

The **veriuser.c** file is located in the following directory:

<path to Quartus II installation>/**eda**/**cadence**/**verilog-xl**

10. Click **Next**.
11. In the **PLI 1.0 Application** page, click **Browse** under **PLI Source Files** to locate the **convert_hex2ver.c** file.
12. Click **Next**.
13. In the **Select Compiler** page, choose your **C** compiler from the **Select Compiler list** box.

    gcc is an example of a C compiler. To allow the **PLIWIZ** wizard to find your C compiler, ensure your path variable is set correctly.

14. Click **Next**.
15. Click **Finish**.
16. To build your targets now, click **Yes**.
17. Compilation creates the file **libpli.so** (**libpli.dll** for PCs), which is your PLI dynamic library, in your session directory. When you elaborate your design, the elaborator looks through the path specified in the `LD_LIBRARY_PATH` (UNIX) or `PATH` (PCs) environment variable, searches for the **.so** and **.dll** files, and loads them when needed.

You must modify `LD_LIBRARY_PATH` or `PATH` to include the directory location of your **.so** and **.dll** files.

Dynamically Load a PLI Library

To create a PLI library to be loaded with the NC-Sim software, perform the following steps:

1. Open the **veriuser.c** file located in the following directory:

   <path to Quartus II installation>/**eda**/**cadence**/**verilog-xl**

   The following two examples are sections of the original and modified **veriuser.c** file. The first example is the original **veriuser.c** file packaged with the Quartus II software. The second example is a **veriuser.c** file modified for dynamic loading.

Original veriuser.c File

```
s_tfcell veriusertfs[] =
{
    /*** Template for an entry:
    { usertask|userfunction, data,
      checktf(), sizetf(), calltf(), misctf(),
      "$tfname", forwref?, Vtool?, ErrMsg? },
    Example:
    {usertask, 0, my_check, 0, my_func, my_misctf, "$my_task" },
    ***/
/*** add user entries here ***/
    /* This Handles Binary bit patterns */
    {usertask, 0, 0, 0, convert_hex2ver, 0, "$convert_hex2ver",
1},

        {0} /*** final entry must be 0 ***/
};
```

Modified veriuser.c File

```
p_tfcell my_bootstrap ()
          {

static s_tfcell my_tfs[] =
/*s_tfcell veriusertfs[] = */
{
    /*** Template for an entry:
    { usertask|userfunction, data,
      checktf(), sizetf(), calltf(), misctf(),
      "$tfname", forwref?, Vtool?, ErrMsg? },
    Example:
    { usertask, 0, my_check, 0, my_func, my_misctf, "$my_task" },
    ***/
/*** add user entries here ***/
    /* This Handles Binary bit patterns */
    {usertask, 0, 0, 0, convert_hex2ver, 0, "$convert_hex2ver",
1},

        {0} /*** final entry must be 0 ***/
};
return(my_tfs);
          }
```

2. Run the PLI wizard by typing `pliwiz` at a command prompt, or on the Utilities menu by clicking **PLI Wizard** in the NCLaunch window.

3. In the **Config Session Name and Directory** page, type the name of the session in the **Config Session Name** box and type the directory in which the file should be built in the **Config Session Directory** box.

4. Click **Next**.

5. In the **Select Simulator/Dynamic Libraries** page, select the **Dynamic Libraries Only** option.

6. Click **Next**.

7. In the **Select Components** page, turn on the **PLI 1.0 Applications** option, and select **loadpli1**.

8. Click **Next**.

9. Type a name into the **Bootstrap Function(s)** box.

   For example, type `my_bootstrap` into the **Bootstrap Function(s)** box.

10. Type the name of your generated dynamic library into the **Dynamic Library** box.

    For example, type `convert_dyn_lib` into the **Dynamic Library** box to generate a dynamic library named **convert_dyn_lib.so**.

11. In the **PLI 1.0 Application** page, click **Browse** under **PLI Source Files** to locate the **convert_hex2ver.c** file and the modified **veriuser.c** file.

12. Click **Next**.

13. In the **Select Compiler** page, select your **C** compiler from the **Select Compiler** list box.

    gcc is an example of a C compiler. To allow the **PLIWIZ** wizard to find your C compiler, ensure your Path variable is set correctly.

14. Click **Next**.

15. Click **Finish**.

16. To build your targets now, click **Yes**.

Compilation generates your dynamic library, **cmd_file.nc**, and **cmd_file.xl** files into your local directory. The **cmd_file.nc** and **cmd_file.xl** files contain command line options to use with your newly generated dynamic library file.

- Use the **cmd_file.nc** command file with **ncelab** to perform your simulations, as shown in the following example:

```
ncelab worklib.mylpmrom -FILE cmd_file.nc ↵
```

- Use the **cmd_file.xl** command file with **verilog-xl** or **ncverilog** to perform your simulations, as shown in the following example:

```
ncverilog -f cmd_file.xl ↵
verilog -f cmd_file.xl ↵
```

Statically Link the PLI Library with NC-Sim

To statically link the PLI library with NC-Sim software, perform the following steps:

1. Run the PLI wizard by typing `pliwiz` at the command prompt, or on the Utilities menu by clicking **PLI Wizard** in the NCLaunch window.
2. In the **Config Session Name** and **Directory** page, type the name of the session in the **Config Session Name** box and type the directory in which the file should be built in the **Config Session Directory** box.
3. Click **Next**.
4. Select **NC Simulators** and select **NC-verilog**.
5. Click **Next**.
6. In the **Select Components** page, turn on the **PLI 1.0 Applications option** and select **Static**.
7. In the **Select PLI 1.0 Application Input** page, select **Existing VERIUSER** (source/object file).

8. Select **Source File** and click **Browse** to locate the **veriuser.c** file provided with the Quartus II software.

   The **veriuser.c** file is found in the following location:

   <path to Quartus II installation>/**eda**/**cadence**/**verilog-xl**

9. Click **Next**.

10. In the **PLI 1.0 Application** page, click **Browse** under **PLI Source Files** to locate the **convert_hex2ver.c** file.

11. Click **Next**.

12. In the **Select Compiler** page, select your **C** compiler from the **Select Compiler** list box.

    gcc is an example of a C compiler. To allow the **PLIWIZ** to find your C compiler, ensure your Path variable is set correctly.

13. Click **Next**.

14. Click **Finish**.

15. To build your targets now, click **Yes**.

Compilation generates **ncelab** and **ncsim** executables into your local directory. These executables replace the original **ncelab** and **ncsim** executables.

ncverilog users can use the following command to perform simulation with the newly generated **ncelab** and **ncsim** executables.

```
ncverilog +ncelabexe+<path to ncelab> +ncsimexe+<path to ncelab> <design files> ↵
```

The following example shows how an **ncverilog** users can perform a simulation with the newly generated **ncelab** and **ncsim** executables:

```
ncverilog +ncelabexe+./ncelab +ncsimexe+./ncsim my_ram.vt my_ram.v -v altera_mf.v ↵
```

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Scripting Reference Manual* includes the same information in PDF format.



For more information about Tcl scripting, refer to the *Tcl Scripting chapter* in volume 2 of the *Quartus II Handbook.* Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Generate NC-Sim Simulation Output Files

You can generate Verilog Output File and Standard Delay Format Output File simulation output files with Tcl commands or at a command prompt.

For more information about generating Verilog Output File simulation output files and Standard Delay Format Output File simulation output files, refer to "Quartus II Simulation Output Files" on page 4–22.

Tcl Commands:

The following three assignments cause a Verilog HDL netlist to be written out when you run the Quartus II netlist writer. The netlist has a 1 ps timing resolution for the NC-Sim Simulation software.

```
set_global_assignment -name EDA_OUTPUT_DATA_FORMAT VERILOG -section_id eda_simulation
set_global_assignment -name EDA_TIME_SCALE "1 ps" -section_id eda_simulation
set_global_assignment -name EDA_SIMULATION_TOOL "NC-Verilog (Verilog)"
```

Use the following Tcl command to run the Quartus II netlist writer:

```
execute_module -tool eda
```

Command Prompt

Use the following command to generate a simulation output file for the Cadence NC-Sim software simulator. Specify Verilog HDL or VHDL for the format.

```
quartus_eda <project name> --simulation --format=<verilog|vhdl> --tool=ncsim ↵
```

Conclusion

The Cadence NC family of simulators work within an Altera FPGA design flow to perform functional/RTL, post-synthesis, and gate-level timing simulation, easily and accurately.

Altera provides functional models of LPM and Altera-specific megafunctions that you can compile with your testbench or design. For timing simulation, use the atom netlist file generated by Quartus II compilation.

The seamless integration of the Quartus II software and Cadence NC tools make this simulation flow an ideal method for fully verifying an FPGA design.

Referenced Documents

This chapter references the following documents:

- *SDF Annotate Guide* and *Cadence NC-Sim User Manual* from Cadence
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 4–10 shows the revision history for this chapter.

Table 4–10. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Updated Table 4–1. • Updated "Operating Condition Example: Generate All Timing Models for Stratix III and Cyclone III Devices" on page 4–26.	Updated for the Quartus II software version 7.2.
May 2007 v7.1.0	• Updated "Software Requirements" on page 4–1. • Updated "Generating a Gate-Level Timing Simulation Netlist" on page 4–24. • Added "Perform Timing Simulation Using Post-Synthesis Netlist" on page 4–27. • Updated "Pulse Reject Delays" on page 4–37. • Updated "Performing a Gate Level Simulation Using NativeLink" on page 4–41. • Updated procedure in "Setting Up a Testbench" on page 4–41. • Added "Referenced Documents" on page 4–51.	—
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—

Table 4–10. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
November 2006 v6.1.0	• Added new software versions to Table 4-1. • Several other minor changes.	Updated for the Quartus II software version 6.1.
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0: • Added a section about setting VCS as the Simulation Tool • Updated EDA Tools Settings in the GUI. • Updated the Synopsys Design Constraints File information. • Added pulse_e and pulse_r information to simulation sections. • Added Quartus II-Generated Testbench information • Updated megafunction information.	—
December 2005 v5.1.1	• Removed reference to convert_hex2ver.obj.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
May 2005 v5.0.0	• Updated information. • Added Using NativeLink with NC-Sim section. • New functionality for Quartus II software 5.0.	—
December 2004 v3.0	Reorganized chapter and updated information.	—
August 2004 v2.1	• New functionality for Quartus II software 4.1 SP1.	—
June 2004 v2.0	• Updates to tables and figures. • New functionality for Quartus II software 4.1.	—
February 2004 v1.0	Initial release.	—

5. Simulating Altera IP in Third-Party Simulation Tools

QII53014-7.2.0

Introduction

The capacity and complexity of Altera® FPGAs continues to increase as the need for intellectual property (IP) becomes increasingly critical. Using IP megafunctions reduces the design and verification time, allowing you to focus on design customization. Altera and the Altera Megafunction Partners Program (AMPP^{SM}) offer a broad portfolio of IP megafunctions optimized for Altera FPGAs. Through parameterization, these reusable blocks of IP can be customized to meet your design requirements.

Even when the IP source code is encrypted or otherwise restricted, Altera's Quartus® II software allows you to easily simulate designs that contain Altera IP. With the Quartus II software, you can custom configure IP designs, then generate a VHDL or Verilog HDL functional simulation model to use with your choice of simulation tools.

This chapter provides an overview of the process for instantiating the IP megafunctions in your design and simulating its' functional simulation model in an Altera-supported, third-party simulation tool. In this chapter, IP megafunctions refer to Altera megafunctions, IP MegaCore® functions and IP AMPP megafunctions. All IP MegaCore functions come with IP functional simulations (IPFS) models to support functional simulation. Some Altera megafunctions and some AMPP megafunctions also require IPFS models for functional simulation.

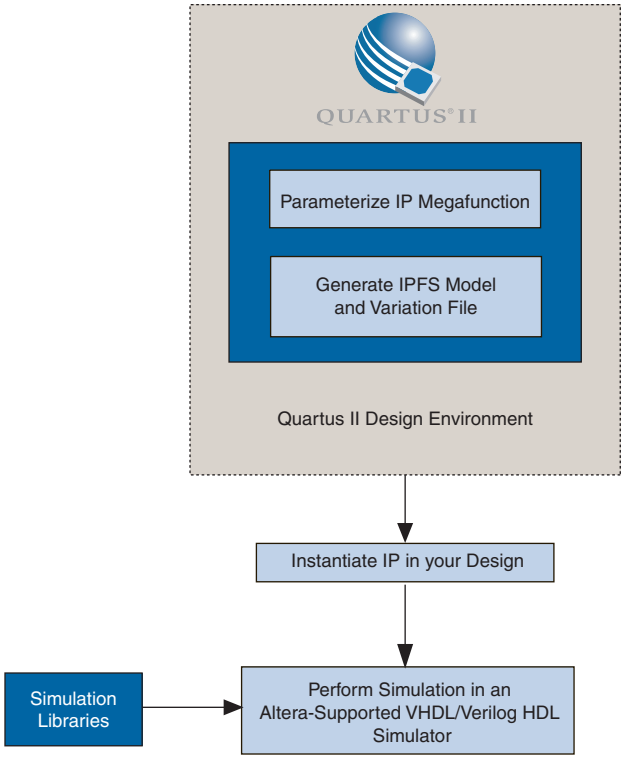
IP Functional Simulation Flow

The IP megafunction's MegaWizard® interface allows you to quickly and easily view documentation, specify parameters, generate an IP functional simulation (IPFS) model, and output the files necessary to integrate a parameterized IP megafunction into your design. Within the Quartus II software, the MegaWizard Plug-In Manager can be used to select and parameterize your choice of IP megafunctions. The Quartus II software generates an IP megafunction's variation file that is included in your Quartus II project. For IP megafunctions that require IPFS models, Quartus II software can also generate a Verilog Output File (**.vo**) or VHDL Output File (**.vho**) that contains a Register Transfer Level (RTL) IPFS model after you have parameterized the megafunction. IPFS models are written to the Quartus II project directory.

Most Altera megafunctions and IP MegaCore functions support functional simulation in Verilog and VHDL for all Altera supported third-party simulators. Simulation libraries are required to simulate IP megafunctions. Refer to Table 5–2 on page 5–10 for a subset of simulation libraries supplied with the Quartus II software.

Figure 5–1 shows a typical simulation flow for Altera IP with third-party simulators.

Figure 5–1. IP Functional Simulation (IPFS) Model Design Flow



Verilog and VHDL IP Functional Simulation (IPFS) Models

Some IP megafunctions require IPFS models to support functional simulation. These IPFS models are written in high-level Register Transfer Level (RTL) HDL. These high-level RTL models in Verilog or VHDL format differ from the low-level synthesized netlist in Verilog or VHDL format generated by the Quartus II software for post-synthesis or post place-and-route simulations. The IPFS models generated by the

Quartus II software are much faster than the low-level post-synthesis or post place-and-route netlists of your design because they are mapped to higher-level primitives such as adders, multipliers, and multiplexers. These IPFS models can be simulated together with the rest of your design in any Altera-supported simulator. Altera recommends that you generate IPFS models in the same hardware language as the IP megafunction's variation file hardware language.



You can use an IPFS model for simulation only, and not for synthesis or any other purpose. Attempting to synthesize an IPFS model will result in a nonfunctional design.



Generating an IPFS model for Altera MegaCore functions does not require a license. However, generating an IPFS model for AMPP megafunctions may require a license. For more information on licensing requirements, contact the IP megafunction vendor.

For details about how to parameterize and generate an IP, refer to the applicable IP user guide.

Instantiate the IP in Your Design

For each IP megafunction in your design, you must instantiate the corresponding entity or module in your design. Each IP megafunction entity or module name is defined in its Quartus II generated megafunction variation file. After instantiating the IP megafunction in your design, you do not need to edit your design for synthesis or simulation.

To synthesize your design using the Quartus II software, add the Quartus II-generated Verilog HDL or VHDL variation file to your Quartus II project. When you create new variation files for a Quartus II project, they are added to the current open project when the megafunction is generated.

To synthesize your design using a third-party EDA tool, add the Quartus II-generated CMP file (<*megafunction variation*>**.cmp**) for your VHDL design or the Verilog HDL black box file (<*megafunction variation*>**_bb.v**) for your Verilog HDL design to your third-party synthesis project.

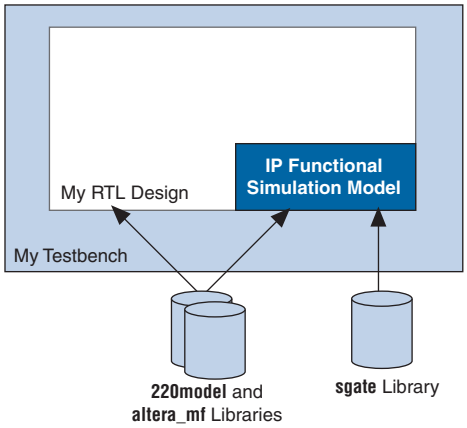


For more information about synthesis and compilation with the Quartus II software, refer to the applicable chapters in volume 1 of the *Quartus II Handbook*.

Perform Simulation

To perform simulation, in addition to adding your design files and testbench files, you also have to add the IP megafunction's variation file or IPFS model to your simulation project. If the IP megafunction does not require an IPFS model for simulation, add the megafunctions' variation file to your simulation project. If the IP megafunction you are simulating requires an IPFS model, then add the IPFS model to your simulation project. Your simulation project will also require Altera-supplied libraries for successful simulation. Figure 5–2 shows how the Altera libraries are used in IP functional simulation.

Figure 5–2. IP Functional Simulation Library Usage



The Quartus II software contains all the libraries required for setting up and running a successful simulation of Altera IP. You can use the Quartus II NativeLink feature to set up your simulation if the IP megafunction you are using supports Quartus II NativeLink. Refer to the applicable IP megafunction user guide to determine if the IP megafunction supports the NativeLink feature in the Quartus II software. Alternatively, you can simulate Altera IP with third-party simulators directly.

Simulating Altera IP Using the Quartus II NativeLink Feature

The Quartus II NativeLink feature eases the task of setting up and running a simulation. The NativeLink feature lets you launch the third-party simulator to perform simulation from within the Quartus II software. The NativeLink feature automates the compilation and simulation of testbenches.

The following list briefly describes the steps to simulate IP megafunctions with third-party simulators using the Quartus II NativeLink feature. Each of these steps is described in more detail in the sections that follow.

1. Set up a Quartus II Project.
2. Select the Third-Party Simulation Tool.
3. Specify the Path for the Third-Party Simulator.
4. Specify the Testbench Settings.
5. Analyze and Elaborate the Quartus II Project.
6. Run RTL Functional Simulation.

Set up a Quartus II Project

To simulate IP megafunctions with the Quartus II NativeLink feature, you must open an existing project or create a new project in the Quartus II software. You can create and parameterize the IP you want to use in your design using MegaWizard Plug-In Manager within the Quartus II software. Altera IP megafunction variation files are added to your Quartus II project when you create and parameterize the IP. You can also add any other required design files to your Quartus II project. If you are using the Quartus II NativeLink feature and your Quartus II project contains IP megafunctions that require IPFS models for simulation, you do not have to manually add the IPFS models to the Quartus II project for these IP megafunctions. When the Quartus II NativeLink feature launches the third-party simulator tool and starts the simulation, it automatically adds the IPFS model files required for simulation as long as they are present in the Quartus II project directory.

Select the Third-Party Simulation Tool

You can select the third-party simulation tool from the Project Settings menu, as shown in Figure 5–3.

Figure 5–3. Selecting Third-Party Simulator Tools

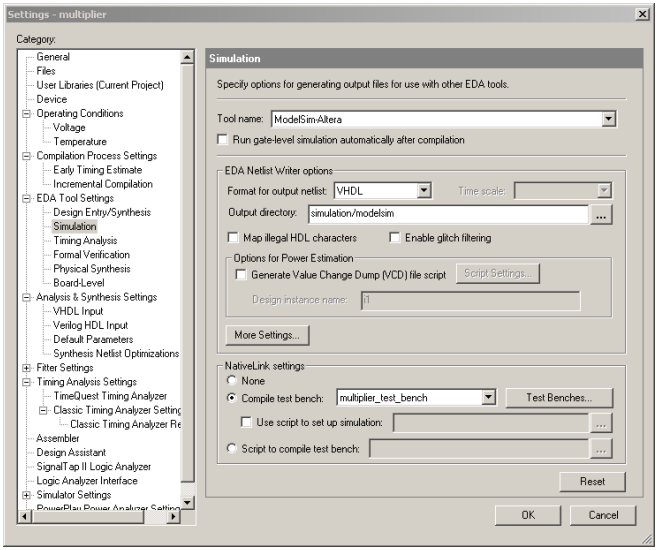


Table 5–1 lists the third-party simulators supported by the Quartus II NativeLink feature.

Table 5–1. Third-Party Simulator Support with the Quartus II NativeLink Feature

Third-Party Simulator	Can be Launched from Quartus II	Testbench Support	Mixed Design (Verilog and VHDL)
ModelSim PE/SE	Yes	Yes	Yes
ModelSim Altera Edition	Yes	Yes	No
Synopsys VCS	Yes *(1)*	Yes	No
Synopsys VCS-MX	Yes *(1)*	Yes	Yes
Cadence NC-Sim	Yes *(1)*	Yes	Yes
Aldec	Yes	Yes	Yes

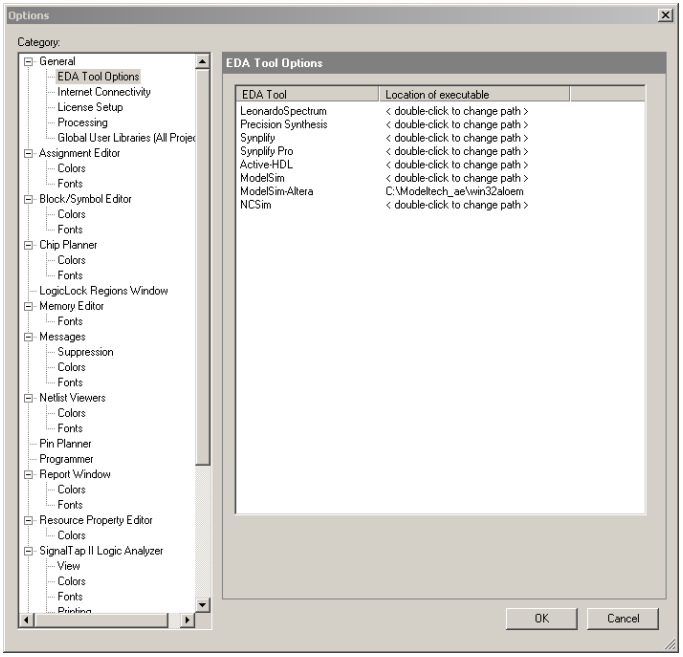
Note to Table 5–1:

(1) If the simulator is run on UNIX or Linux platforms, the Quartus II software must be running on the same platform to launch the simulator tool.

Specify the Path for the Third-Party Simulator

To launch the third-party simulation tool, the absolute path for the selected simulator must be provided in the **Options** page under the Tools menu. See Figure 5–4. Double click the **Location of executable** field to change or specify the absolute path.

Figure 5–4. Specifying the Simulator Path

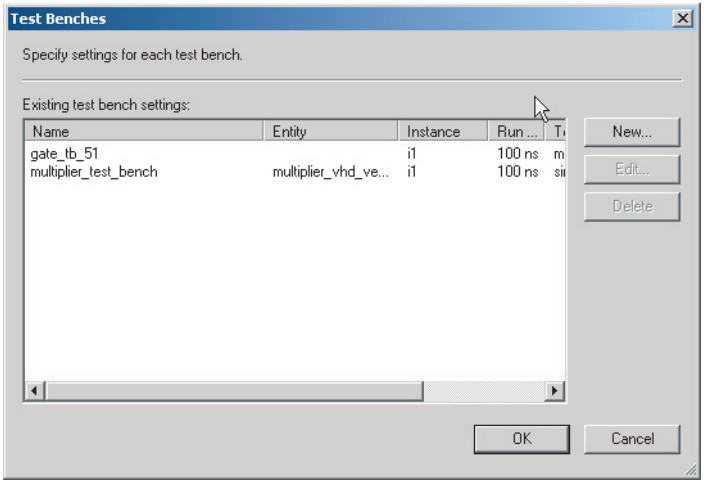


Specify the Testbench Settings

Specify the applicable testbench settings as follows:

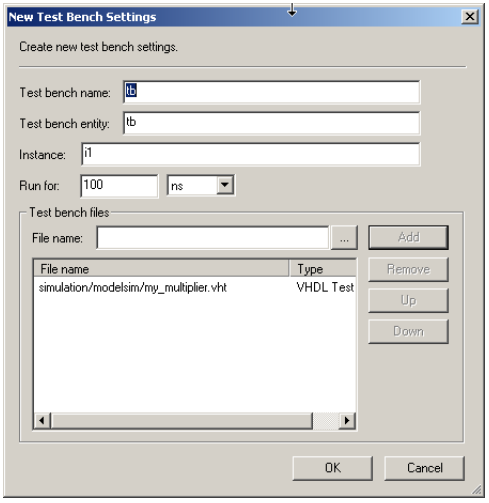
1. Under the NativeLink Settings in the **Settings** dialog box (Figure 5–3), select the **Compile Test Bench** radio button and click **Test Benches** to display the **Test Benches** dialog box. See Figure 5–5.

Figure 5–5. Test Bench Dialog Box



2. Click **New** to display the **New Test Bench Settings** dialog box (shown in Figure 5–6).

Figure 5–6. New Test Bench Settings Dialog Box



3. In the **New Test Bench Settings** dialog box, set the appropriate fields with the names for the testbenches.

For specific instructions about specifying testbench settings for your MegaCore function, refer to your MegaCore function user guide.

4. After specifying the testbench files, close the **New Test Bench Settings**, **Test Benches**, and **Settings** dialog boxes.

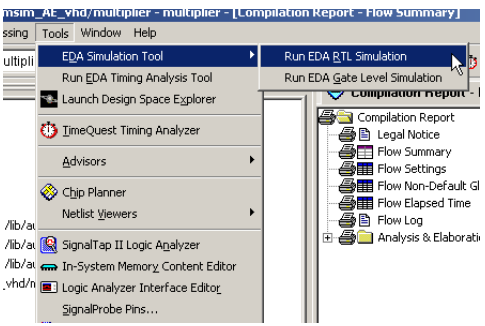
Analyze and Elaborate the Quartus II Project

Before starting the simulation using the NativeLink feature, make sure that each IP megafunctions' variation files are included in your design project. On the Quartus II Processing menu, point to Start, then click **Start Analysis & Elaboration.**

Run RTL Functional Simulation

After the design is analyzed and elaborated, you can start the simulation by clicking **Run EDA-RTL Simulation** from the Tools menu. See Figure 5–7. During RTL functional simulation, the IPFS models are compiled and used by the simulator.

Figure 5–7. Running Functional Simulation for IP using NativeLink



Simulating Altera IP Without the Quartus II NativeLink Feature

You can also simulate Altera IP directly with third-party simulators. If your design instantiates an IP megafunction, add its variation file to your simulation project. If the IP megafunction requires IPFS model files, **do not** add the megafunctions' variation file to your simulation project. Rather, add its' IPFS model files (either Verilog or VHDL) to your simulation project. The IPFS model generated by the Quartus II software instantiates high-level primitives such as adders, multipliers, and multiplexers, as well as the library of parameterized modules (LPM) functions and Altera megafunctions.

To properly compile, load, and simulate the IP megafunctions, you must first compile the following libraries in your simulation tool:

- **sgate**—includes the definition of the high-level primitives (needed for IPFS models)
- **altera_mf**—includes the definition of Altera megafunctions
- **220model**—includes the definition of LPM functions

You can use these library files with any Altera-supported simulation tool. If you are using the ModelSim® Altera software, the libraries are precompiled and mapped.



To simulate a design containing a Nios® processor or Avalon® peripherals, refer to *AN 189 Simulating Nios Embedded Processor Designs*.

Table 5–2 lists the simulation library files, where <*path*> is the directory where the Quartus II software is installed.

Table 5–2. Simulation Library Files

Location	HDL Language	Description
<*path*>**/eda/sim_lib/sgate.v**	Verilog HDL	Libraries that contain simulation models for IP functional models
<*path*>**/eda/sim_lib/sgate.vhd**	VHDL	
<*path*>**/eda/sim_lib/sgate_pack.vhd**	VHDL	Libraries that contain VHDL component declarations for the **sgate.vhd** library
<*path*>**/eda/sim_lib/220model.v**	Verilog HDL	Libraries that contain simulation models for the Altera LPM version 2.2.0
<*path*>**/eda/sim_lib/220model.vhd**	VHDL	
<*path*>**/eda/sim_lib/220pack.vhd**	VHDL	Libraries that contain VHDL component declarations for the **220model.vhd** library
<*path*>**/eda/sim_lib/altera_mf.v**	Verilog HDL	Libraries that contain simulation models for Altera-specific megafunctions
<*path*>**/eda/sim_lib/altera_mf.vhd**	VHDL	
<*path*>**/eda/sim_lib/altera_mf_components.vhd**	VHDL	Libraries that contain VHDL component declarations for the **altera_mf.vhd** library

Design Language Examples

The following design language examples explain how to simulate IP megafunctions directly with third-party simulator tools. These design examples describe simulation with:

- ModelSim Verilog
- ModelSim VHDL
- NC-VHDL
- VCS

Verilog HDL Example: Simulating the IPFS Model in the ModelSim Software

The following example shows the process of simulating a Verilog HDL-based megafunction. The example assumes that the megafunction variation and the IPFS model are generated.

1. Create a ModelSim project by performing the following steps:
   a. In the ModelSim software, on the File menu, point to New and click **Project**. The **Create Project** dialog box is shown.
   b. Specify the name of your simulation project.
   c. Specify the desired location for your simulation project.
   d. Specify the default library name and click **OK**.
   e. Add relevant files to your simulation project:
      - Your design files
      - The IPFS model generated by the Quartus II software (if you are using the ModelSim-Altera software, skip to step 5)
      - The **sgate.v**, **220model.v**, and **altera_mf.v** library files
2. Create the required simulation libraries by typing the following commands at the ModelSim prompt:

```
vlib sgate ↵

vlib lpm ↵

vlib altera_mf ↵
```

3. Map to the required simulation libraries by typing the following commands at the ModelSim prompt:

```
vmap sgate sgate ↵

vmap lpm lpm ↵

vmap altera_mf altera_mf ↵
```

4. Compile the HDL into libraries by typing the following commands at the ModelSim prompt:

```
vlog -work altera_mf altera_mf.v ↵

vlog -work sgate sgate.v ↵

vlog -work lpm 220model.v ↵
```

5. Compile the IPFS model by typing the following command at the ModelSim prompt:

```
vlog -work work <my_IP>.vo ↵
```

6. Compile your RTL by typing the following command at the ModelSim prompt:

```
vlog -work work <my_design>.v ↵
```

7. Compile the testbench by typing the following command at the ModelSim prompt:

```
vlog -work work <my_testbench>.v ↵
```

8. Load the testbench by typing the following command at the ModelSim prompt:

```
vsim -L <altera_mf library_path> -L <lpm_library_path>
-L <sgate_library_path> work.<my_testbench> ↵
```

VHDL Example: Simulating the IPFS Model in the ModelSim Software

The following example shows the process of performing a functional simulation of a VHDL-based, megafunction IPFS model. The example assumes that the megafunction's variation and the IPFS model are generated.

1. Create a ModelSim project by performing the following steps:
   a. In the ModelSim software, on the File menu, point to New and click **Project**. The **Create Project** dialog box appears.
   b. Specify the name for your simulation project.
   c. Specify the desired location for your simulation project.
   d. Specify the default library name and click **OK**.
   e. Add the relevant files to your simulation project:
      - Add your design files
      - Add the IPFS model generated by the Quartus II software (if you are using the ModelSim-Altera software, skip to step 5)
      - Add the **sgate.vhd**, **sgate_pack.vhd**, **220model.vhd**, **220pack.vhd**, **altera_mf.vhd**, and **altera_mf_components.vhd** library files
2. Create the required simulation libraries by typing the following commands at the ModelSim prompt:

```
vlib sgate ↵

vlib lpm ↵

vlib altera_mf ↵
```

3. Map to the required simulation libraries by typing the following commands at the ModelSim prompt:

```
vmap sgate sgate ↵

vmap lpm lpm ↵

vmap altera_mf altera_mf ↵
```

4. Compile the HDL into libraries by typing the following commands at the ModelSim prompt:

```
vcom -work altera_mf -93 -explicit
altera_mf_components.vhd ↵

vcom -work altera_mf -93 -explicit altera_mf.vhd ↵
```

```
vcom -work lpm -93 -explicit 220pack.vhd ↵

vcom -work lpm -93 -explicit 220model.vhd ↵

vcom -work sgate -93 -explicit sgate_pack.vhd ↵

vcom -work sgate -93 -explicit sgate.vhd ↵
```

5. Compile the IPFS model by typing the following command at the ModelSim prompt:

```
vcom -work work -93 -explicit <output_netlist>.vho ↵
```

6. Compile the RTL by typing the following command at the ModelSim prompt:

```
vcom -work work -93 -explicit <RTL>.vhd ↵
```

7. Compile the testbench by typing the following command at the ModelSim prompt:

```
vcom -work work -93 -explicit <my_testbench>.vhd ↵
```

8. Load the testbench by typing the following command at the ModelSim prompt:

```
vsim work.my_testbench ↵
```

NC-VHDL Example: Simulating the IPFS Model in the NC-VHDL Software

The following example shows the process of performing a functional simulation of an NC-VHDL-based, megafunction IP functional-simulation model. The example assumes that the megafunction's variation and the IPFS model are generated.

1. Create a **cds.lib** file by typing the following entries:

```
DEFINE worklib ./worklib

DEFINE sgate ./sgate

DEFINE altera_mf ./altera_mf

DEFINE lpm ./lpm
```

2. Compile library files into appropriate libraries by typing the following commands at the command prompt:

```
ncvhdl –V93 –WORK lpm 220pack.vhd ↵

ncvhdl –V93 –WORK lpm 220model.vhd ↵

ncvhdl –V93 –WORK altera_mf
altera_mf_components.vhd ↵

rncvhdl –V93 –WORK altera_mf altera_mf.vhd ↵

ncvhdl –V93 –WORK sgate sgate_pack.vhd ↵

ncvhdl –V93 –WORK sgate sgate.vhd ↵
```

3. Compile source code and testbench files by typing the following commands at the command prompt:

```
ncvhdl –V93 –WORK worklib <my_design>.vhd ↵

ncvhdl –V93 –WORK worklib <my_testbench>.vhd ↵

ncvhdl –V93 –WORK worklib
<my_IPtoolbench_output_netlist>.vho ↵
```

4. Elaborate the design by typing the following command at the command prompt:

```
ncelab worklib.<my_testbench>:entity ↵
```

Verilog HDL Example: Simulating Your IPFS Model in VCS

The following example illustrates the process of performing a functional simulation of a design that contains a Verilog HDL-based, megafunction IPFS model. This example assumes that the megafunction variation and the IPFS model are generated.

Single-Step Process

For the single-step process, type the following at the command prompt:

```
vcs <testbench>.v <RTL>.v <output_netlist>.v -v 220model.v
altera_mf.v sgate.v -R ↵
```

Two-Step Process (Compilation and Simulation)

For compilation and simulation, perform the following steps:

1. Compile your design files by typing the following at the command prompt:

   ```
   vcs <testbench>.v <RTL>.v <output_netlist>.v -v 220model.v
   altera_mf.v sgate.v -o simulation_out ↵
   ```

2. Load your simulation by typing the following at a command prompt:

   ```
   source simulation_out ↵
   ```



For more information about simulating a design in VCS, refer to the chapter *Synopsys VCS Support* in volume 3 of the *Quartus II Handbook*.

Conclusion

Altera Quartus II software provides full support for simulating IP megafunction's with third party tools either directly or using its NativeLink feature. Using the Quartus II software, you can also generate IPFS models for supported megafunctions that enhances and simplifies design verification. Using an IPFS model is transparent, requiring only the addition of different files in which to synthesize and simulate projects.

Referenced Documents

This chapter references the following documents:

- *Synopsys VCS Support* in volume 3 of the *Quartus II Handbook*
- Volume 1 of the *Quartus II Handbook*

Document Revision History

Table 5–3 shows the revision history for this chapter.

Table 5–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 5–16.	—
May 2007 v7.1.0	• Updated Figure 5–6 • Added "Referenced Documents" section.	Minor updates to support the Quartus II software, version 7.1.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only.	—
November 2006 v6.1.0	• Added Quartus II NativeLink feature information • Updated Figure 5-1, 5-2 • Added Figure 5-3, 5-4, 5-7 • Added Table 5-1	Chapter updates to support the Quartus II NativeLink feature.
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
May 2005 v5.0.0	Chapter 4 was formerly in Section I, Vol 3 in 4.2.	—
December 2004 v1.0.0	Initial release.	—

Section II. Timing Analysis

As designs become more complex, the need for advanced timing analysis capability grows. Static timing analysis is a method of analyzing, debugging, and validating the timing performance of a design. The Quartus® II software provides the features necessary to perform advanced timing analysis for today's system-on-a-programmable-chip (SOPC) designs.

Synopsys PrimeTime is an industry standard sign-off tool, used to perform static timing analysis on most ASIC designs. The Quartus II software provides a path to enable you to run PrimeTime on your Quartus II software designs, and export a netlist, timing constraints, and libraries to the PrimeTime environment.

This section explains the basic principles of static timing analysis, the advanced features supported by the Quartus II Timing Analyzer, and how you can run PrimeTime on your Quartus designs.

This section includes the following chapters:

- Chapter 6, The Quartus II TimeQuest Timing Analyzer
- Chapter 7, Switching to the Quartus II TimeQuest Timing Analyzer
- Chapter 8, Quartus II Classic Timing Analyzer
- Chapter 9, Synopsys PrimeTime Support



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

6. The Quartus II TimeQuest Timing Analyzer

QII53018-7.2.0

Introduction

The Quartus® II TimeQuest Timing Analyzer is a powerful ASIC-style timing analysis tool that validates the timing performance of all logic in your design using an industry-standard constraint, analysis, and reporting methodology. Use the Quartus II TimeQuest Timing Analyzer's GUI or command-line interface to constrain, analyze, and report results for all timing paths in your design.

Before running the Quartus II TimeQuest Timing Analyzer, you must specify initial timing constraints that describe the clock characteristics, timing exceptions, and signal transition arrival and required times. You can specify timing constraints in the Synopsys Design Constraints (SDC) file format using the GUI or command-line interface. The Quartus II Fitter optimizes the placement of logic to meet your constraints.

During timing analysis, the Quartus II TimeQuest Timing Analyzer analyzes the timing paths in the design, calculates the propagation delay along each path, checks for timing constraint violations, and reports timing results as slack in the **Report** pane and in the **Console** pane. If the Quartus II TimeQuest Timing Analyzer reports any timing violations, you can customize the reporting to view precise timing information about specific paths, and then constrain those paths to correct the violations. When your design is free of timing violations, you can be confident that the logic will operate as intended in the target device.

the Quartus II TimeQuest Timing Analyzer is a complete static timing analysis tool that you can use as a sign-off tool for Altera® FPGAs and structured ASICs.

This chapter contains the following sections:

- "Getting Started with the Quartus II TimeQuest Timing Analyzer"
- "Compilation Flow with the Quartus II TimeQuest Timing Analyzer Guidelines" on page 6–3
- "Timing Analysis Overview" on page 6–7
- "Specify Design Timing Requirements" on page 6–19
- "The Quartus II TimeQuest Timing Analyzer Flow Guidelines" on page 6–22
- "Collections" on page 6–23
- "Constraints Files" on page 6–25
- "Clock Specification" on page 6–28
- "I/O Specifications" on page 6–45

- "Timing Exceptions" on page 6–48
- "Constraint and Exception Removal" on page 6–57
- "Timing Reports" on page 6–58
- "Timing Analysis Features" on page 6–77
- "The TimeQuest Timing Analyzer GUI" on page 6–82
- "Conclusion" on page 6–95

Getting Started with the Quartus II TimeQuest Timing Analyzer

The Quartus II TimeQuest Timing Analyzer caters to the needs of the most basic to the most advanced designs for FPGAs.

This section provides a brief overview of the Quartus II TimeQuest Timing Analyzer, including the necessary steps to properly constrain a design, perform a full place-and-route, and perform reporting on the design.

Setting Up the Quartus II TimeQuest Timing Analyzer

The Quartus II software version 7.2 supports two native timing analysis tools: Quartus II TimeQuest Timing Analyzer and the Quartus II Classic Timing Analyzer. When you specify the Quartus II TimeQuest Timing Analyzer as the default timing analysis tool, the Quartus II TimeQuest Timing Analyzer guides the Fitter and analyzes timing results after compilation.

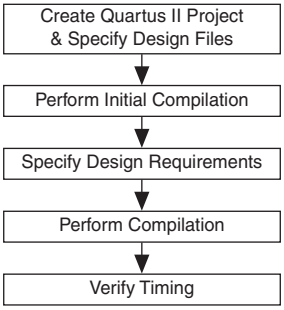
To specify the Quartus II TimeQuest Timing Analyzer as the default timing analyzer, on the Assignments menu, click **Settings**. In the **Settings** dialog box, in the **Category** list, select **Timing Analysis Settings**, and turn on **Use TimeQuest Timing Analyzer during compilation**.

To add the TimeQuest icon to the Quartus II toolbar, on the Tools menu, click **Customize**. In the **Customize** dialog box, click the **Toolbars** tab, turn on **Processing**, and click **Close**.

Compilation Flow with the Quartus II TimeQuest Timing Analyzer Guidelines

When you enable the Quartus II TimeQuest Timing Analyzer as the default timing analyzer, everything from constraint validation to timing verification is performed by the Quartus II TimeQuest Timing Analyzer. Figure 6–1 shows the recommended design flow steps to maximize and leverage the benefits the Quartus II TimeQuest Timing Analyzer. Details about each step are provided after the figure.

Figure 6–1. Design Flow with the Quartus II TimeQuest Timing Analyzer



- **Create Quartus II Project and Specify Design Files**—Creates a project before you can compile design files. In this step you specify the target FPGA, any EDA tools used in the design cycle, and all design files.

  You can also modify existing design files for design optimization and add additional design files. For example, you can add HDL files or schematics to the project.

- **Perform Initial Compilation**—Creates an initial design database before you specify timing constraints for your design. Perform Analysis and Synthesis to create a post-map database, or perform a full compilation to create a post-fit database.

  Creating a post-map database for the initial compilation is faster than creating a post-fit database, and a post-map database is sufficient for the initial database.

  Creating a post-fit database is recommended only if you previously created and specified an SDC file for the project. A post-map database is sufficient for the initial compilation.

- **Specify Design Requirements**—Timing requirements guide the Fitter as it places and routes your design.

  You must enter all timing constraints and exceptions in an SDC file. This file must be included as part of the project. To add this file to your project, on the Project menu, click **Add/Remove Files in Project**, and add the SDC file in the **Files** dialog box.



Refer to "Specify Timing Constraints" on page 6–20 for a list of timing constraints and exceptions.

- **Perform Compilation**—Synthesizes, places, and routes your design into the target FPGA.

  When compilation is complete, the TimeQuest Timing Analyzer generates summary clock setup and clock hold, recovery, and removal reports for all defined clocks in the design.

- **Verify Timing**—Verifies timing in your design with the Quartus II TimeQuest Timing Analyzer

Running the Quartus II TimeQuest Timing Analyzer

You can run the Quartus II TimeQuest Timing Analyzer in one of the following modes:

- Directly from the Quartus II software
- Stand-alone mode
- Command-line mode

This section describes each of the modes, and the behavior of the Quartus II TimeQuest Timing Analyzer.

Directly from the Quartus II Software

To run the Quartus II TimeQuest Timing Analyzer from the Quartus II software, on the Tools menu, click **TimeQuest Timing Analyzer**. The Quartus II TimeQuest Timing Analyzer is available after you have created a database for the current project. The database can be either a post-map or post-fit database; perform Analysis and Synthesis to create a post-map database, or a full compilation to create a post-fit database.



After a database is created in the Quartus II software, you can create a timing netlist based on that database. If you create a post-map database, you cannot create a post-fit timing netlist in the Quartus II TimeQuest Timing Analyzer.

When you launch the TimeQuest Timing Analyzer directly from the Quartus II software, the current project opens by default.

Stand-Alone Mode

To run the Quartus II TimeQuest Timing Analyzer in stand-alone mode, type the following command at the command prompt:

```
quartus_staw ↵
```

In stand-alone mode, you can perform static analysis on any project that contains either a post-map or post-fit database. To open a project, double-click **Open Project** in the **Tasks** pane.

Command-Line Mode

Use the command-line mode for easy integration with scripted design flows. Using the command-line mode avoids interaction with the user interface provided by the Quartus II TimeQuest Timing Analyzer, but allows the automation of each step of the static timing analysis flow. Table 6–1 provides a summary of the options available in the command-line mode.

Table 6–1. Summary of Command Line Options (Part 1 of 2)

Command Line Option	Description
`-h | --help`	Provides help information on `quartus_sta`.
`-t` <*script file*> `|` `--script=`<*script file*>	Sources the <*script file*>.
`-s | --shell`	Enters shell mode.
`--tcl_eval` <*tcl command*>	Evaluates the Tcl command <*tcl command*>.
`--do_report_timing`	Runs the command: `report_timing -npaths 1 -to_clock $clock` for all clocks in the design.
`--force_dat`	Forces the Delay Annotator to annotate the new delays from the recently compiled design to the compiler database.
`--lower_priority`	Lowers the computing priority of the `quartus_sta` process.
`--post_map`	Uses the post-map database results.
`--qsf2sdc`	Converts assignments from the Quartus II Settings File (**.qsf**) format to the Synopsys Design Constraints File format.
`--sdc=`<*SDC file*>	Specifies the SDC file to read.
`--fast_model`	Uses the fast corner delay models.
`--report_script=`<*script*>	Specifies a custom report script to be called.
`--speed=`<*value*>	Specifies the device speed grade to be used for timing analysis.

Table 6–1. Summary of Command Line Options (Part 2 of 2)

Command Line Option	Description
`--tq2hc`	Generate temporary files to convert the Quartus II TimeQuest Timing Analyzer SDC file(s) to a PrimeTime SDC file that can be used by the HardCopy Design Center (HCDC).
`--tq2pt`	Generate temporary files to convert the Quartus II TimeQuest Timing Analyzer SDC file(s) to a PrimeTime SDC file.
`-f` *<argument file>*	Specifies a file containing additional command-line arguments.
`-c` *<revision name>* | `--rev=`*<revision_name>*	Specifies which revision and its associated Quartus II Settings File (**.qsf**) to use.
`--multicorner`	Specifies that all slack summary reports be generated for both the slow and fast corners.

To run the Quartus II TimeQuest Timing Analyzer in command-line mode, type the following command at the command prompt:

`quartus_sta` *<options>* ↵

Timing Analysis Overview

This section provides an overview of the Quartus II TimeQuest Timing Analyzer concepts. Understanding these concepts allows you to take advantage of the powerful timing analysis features available in the Quartus II TimeQuest Timing Analyzer.

the Quartus II TimeQuest Timing Analyzer follows the flow shown in Figure 6–2 when it analyzes your design. Table 6–2 lists the most commonly used commands for each step.

Figure 6–2. The the Quartus II TimeQuest Timing Analyzer Flow

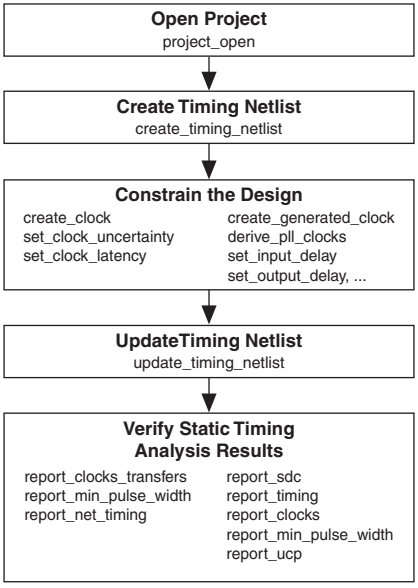


Table 6–2 describes the Quartus II TimeQuest Timing Analyzer terminology.

Table 6–2. The Quartus II TimeQuest Timing Analyzer Terms

Terminology	Definition
Nodes	Most basic timing netlist unit. Use to represent ports, pins, registers, and keepers.
Keepers	Ports or registers. *(1)*
Cells	Look-up table (LUT), registers, DSP blocks, TriMatrix® memory, IOE, and so on. *(2)*
Pins	Inputs or outputs of cells.
Nets	Connections between pins.
Ports	Top-level module inputs or outputs; for example, device pins.
Clocks	Abstract objects outside of the design.

Notes to Table 6–2:

(1) Pins can indirectly refer to keepers. For example, when the value in the `-from` field of a constraint is a clock pin to a dedicated memory. In this case, the clock pin refers to a collection of registers.

(2) For Stratix® devices and other early device families, the LUT and registers are contained in logic elements (LE) and act as cells for these device families.

The Quartus II TimeQuest Timing Analyzer requires a timing netlist before it can perform a timing analysis on any design. For example, for the design shown in Figure 6–3, the Quartus II TimeQuest Timing Analyzer generates a netlist equivalent to the one shown in Figure 6–4.

Figure 6–3. Sample Design

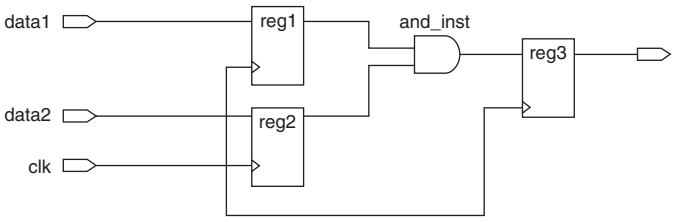


Figure 6–4. The Quartus II TimeQuest Timing Analyzer Timing Netlist

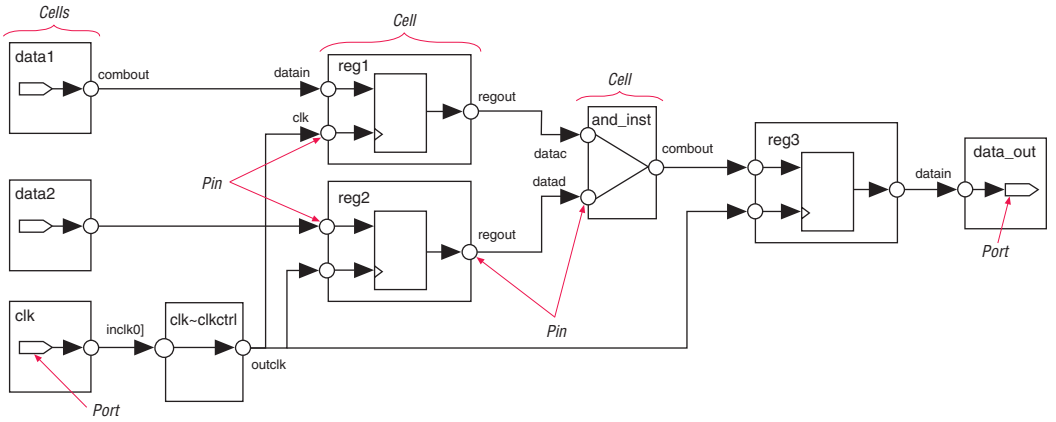


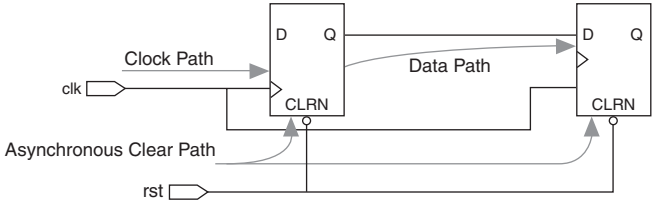
Figure 6–4 shows various cells, pins, nets, and ports. Sample cell names are `reg1`, `reg2`, and `and_inst`; sample pins are `data1|combout`, `reg1|regout`, and `and_inst|combout`; sample net names are `data1~combout`, `reg1`, and `and_inst`; and sample port names are `data1`, `clk`, and `data_out`.

Paths connect two design nodes, such as the output of a register to the input of another register. Timing paths play a significant role in timing analysis. Understanding the types of timing paths is important to timing closure and optimization. The following list shows some of the commonly analyzed paths that are described in this section:

- **Edges**—the connections from ports-to-pins, from pins-to-pins, and from pins-to-ports.
- **Clock paths**—the edges from device ports or internally generated clock pins to the clock pin of a register.
- **Data paths**—the edges from a port or the data output pin of a sequential element to a port or the data input pin of another sequential element.
- **Asynchronous paths**—the edges from a port or sequential element to the asynchronous set or clear pin of a sequential element.

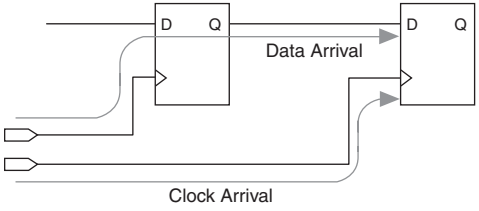
Figure 6–5 shows some of these commonly analyzed path types.

Figure 6–5. Path Types



Once the Quartus II TimeQuest Timing Analyzer identifies the path type, it can report data and clock arrival times for valid register-to-register paths. The Quartus II TimeQuest Timing Analyzer calculates data arrival time by adding the delay from the clock source to the clock pin of the source register, the micro clock-to-out (μt_{CO}) of the source register, and the delay from the source register's *Q* pin to the destination register's *D* pin, where the μt_{CO} is the intrinsic clock-to-out for the internal registers in the FPGA. The Quartus II TimeQuest Timing Analyzer calculates clock arrival time by adding the delay from the clock source to the destination register's clock pin. Figure 6–6 shows a data arrival path and a clock arrival path. The Quartus II TimeQuest Timing Analyzer calculates data required time by accounting for the clock arrival time and the micro setup time (μt_{SU}) of the destination register, where the μt_{SU} is the intrinsic setup for the internal registers in the FPGA.

Figure 6–6. Data Arrival and Clock Arrival

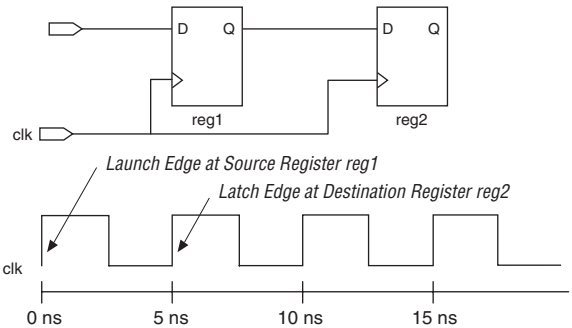


In addition to identifying various paths in a design, the Quartus II TimeQuest Timing Analyzer analyzes clock characteristics to compute the worst-case requirement between any two registers in a single register-to-register path. You should constrain all clocks in your design before performing this analysis.

The launch edge is an active clock edge that sends data out of a sequential element, acting as a source for the data transfer. A latch edge is the active clock edge that captures data at the data port of a sequential element, acting as a destination for the data transfer.

Figure 6–7 shows a single-cycle system that uses consecutive clock edges to transfer and capture data, a register-to-register path, and the corresponding launch and latch edges timing diagram. In this example, the launch edge sends the data out of register `reg1` at 0 ns, and register `reg2` latch edge captures the data at 5 ns.

Figure 6–7. Launch Edge and Latch Edge



The Quartus II TimeQuest Timing Analyzer validates clock setup and hold requirements relative to the launch and latch edges.

Clock Analysis

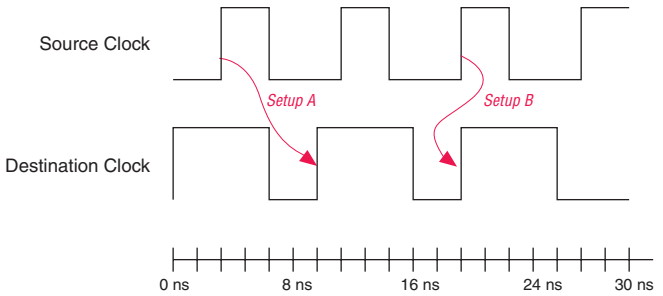
A comprehensive static timing analysis includes analysis of register-to-register, I/O, and asynchronous reset paths. The Quartus II TimeQuest Timing Analyzer uses data required times, data arrival times, and clock arrival times to verify circuit performance and detect possible timing violations. The Quartus II TimeQuest Timing Analyzer determines the timing relationships that must be met for the design to correctly function, and checks arrival times against required times to verify timing.

Clock Setup Check

To perform a clock setup check, the Quartus II TimeQuest Timing Analyzer determines a setup relationship by analyzing each launch and latch edge for each register-to-register path. For each latch edge at the destination register, the Quartus II TimeQuest Timing Analyzer uses the closest previous clock edge at the source register as the launch edge. In

Figure 6–8, two setup relationships are defined and are labeled Setup A and Setup B. For the latch edge at 10 ns, the closest clock that acts as a launch edge is at 3 ns and is labeled Setup A. For the latch edge at 20 ns, the closest clock that acts as a launch edge is 19 ns, and is labeled Setup B.

Figure 6–8. Setup Check



The Quartus II TimeQuest Timing Analyzer reports the result of clock setup checks as slack values. Slack is the margin by which a timing requirement is met or not met. Positive slack indicates the margin by which a requirement is met, and negative slack indicates the margin by which a requirement is not met. The Quartus II TimeQuest Timing Analyzer determines clock setup slack, as shown in Equation 1, for internal register-to-register paths.

(1)

$$\text{Clock Setup Slack} = \text{Data Required Time} - \text{Data Arrival Time}$$

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} + \text{Register-to-Register Delay}$$

$$\text{Data Required} = \text{Clock Arrival Time} - \mu t_{SU} - \text{Setup Uncertainty}$$

$$\text{Clock Arrival Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register}$$

If the data path is from an input port to a internal register, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 2 to calculate the setup slack time.

(2)

$$\text{Clock Setup Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay} + \text{Input Maximum Delay of Pin} + \text{Pin-to-Register Delay}$$

$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} - \mu t_{SU}$$

If the data path is an internal register to an output port, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 3 to calculate the setup slack time.

(3)

$$\text{Clock Setup Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$

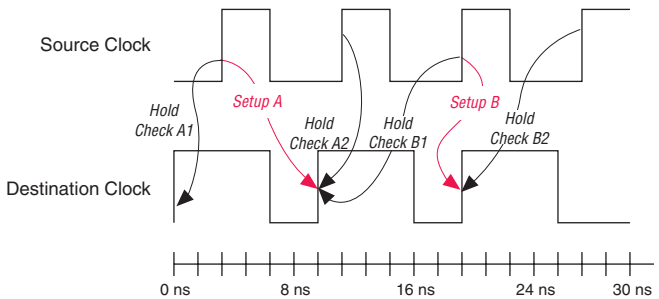
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} + \text{Register-to-Pin Delay}$$

$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay} - \text{Output Maximum Delay of Pin}$$

Clock Hold Check

To perform a clock hold check, the Quartus II TimeQuest Timing Analyzer determines a hold relationship for each possible setup relationship that exists for all source and destination register pairs. The Quartus II TimeQuest Timing Analyzer checks all adjacent clock edges from all setup relationships to determine the hold relationships. The Quartus II TimeQuest Timing Analyzer performs two hold checks for each setup relationship. The first hold check determines that the data launched by the current launch edge is not captured by the previous latch edge. The second hold check determines that the data launched by the next launch edge is not captured by the current latch edge. Figure 6–9 shows two setup relationships labeled setup A and setup B. The first hold check is labeled hold check A1 and hold check B1 for setup A and setup B, respectively. The second hold check is labeled hold check A2 and hold check B2 for setup A and setup B, respectively.

Figure 6–9. Hold Checks



From the possible hold relationships, The Quartus II TimeQuest Timing Analyzer selects the hold relationship that is the most restrictive. The hold relationship with the largest difference between the latch and launch edges (that is, latch – launch and not the absolute value of latch and

launch) is selected because this determines the minimum allowable delay for the register-to-register path. For Figure 6–9, the hold relationship selected is hold check A2.

The Quartus II TimeQuest Timing Analyzer determines clock hold slack as shown in Equation 4.

(4) Clock Hold Slack = Data Arrival Time – Data Required Time

Data Arrival Time = Launch Edge + Clock Network Delay to Source Register + μt_{CO} + Register-to-Register Delay

Data Required Time = Clock Arrival Time + μt_{H} + Hold Uncertainty

Clock Arrival Time = Latch Edge + Clock Network Delay to Destination Register

If the data path is from an input port to an internal register, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 5 to calculate the setup slack time.

(5) Clock Setup Slack Time = Data Arrival Time – Data Required Time

Data Arrival Time = Launch Edge + Clock Network Delay + Input Minimum Delay of Pin + Pin-to-Register Delay

Data Required Time = Latch Edge + Clock Network Delay to Destination Register + μt_{H}

If the data path is an internal register to an output port, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 6 to calculate the setup slack time.

(6) Clock Setup Slack Time = Data Arrival Time – Data Required Time

Data Arrival Time = Latch Edge + Clock Network Delay to Source Register + μt_{CO} + Register-to-Pin Delay

Data Required Time = Latch Edge + Clock Network Delay – Output Minimum Delay of Pin

Recovery and Removal

Recovery time is the minimum length of time the de-assertion of an asynchronous control signal, for example, `clear` and `preset`, must be stable before the next active clock edge. The recovery slack time calculation is similar to the clock setup slack time calculation, but it applies to asynchronous control signals. If the asynchronous control signal is registered, the Quartus II TimeQuest Timing Analyzer uses Equation 7 to calculate the recovery slack time.

(7)

$$\text{Recovery Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} + \text{Register-to-Register Delay}$$

$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} - \mu t_{SU}$$

If the asynchronous control is not registered, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 8 to calculate the recovery slack time.

(8)

$$\text{Recovery Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay} + \text{Maximum Input Delay} + \text{Port-to-Register Delay}$$

$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register Delay} - \mu t_{SU}$$



If the asynchronous reset signal is from a port (device I/O), you must make an Input Maximum Delay assignment to the asynchronous reset port for the Quartus II TimeQuest Timing Analyzer to perform recovery analysis on that path.

Removal time is the minimum length of time the de-assertion of an asynchronous control signal must be stable after the active clock edge. The Quartus II TimeQuest Timing Analyzer removal time slack calculation is similar to the clock hold slack calculation, but it applies asynchronous control signals. If the asynchronous control is registered, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 9 to calculate the removal slack time.

(9)

$$\text{Removal Slack Time} = \text{Data Arrival Time} - \text{Data Required Time}$$

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} \text{ of Source Register} + \text{Register-to-Register Delay}$$

$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} + \mu t_{H}$$

If the asynchronous control is not registered, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 10 to calculate the removal slack time.

(10)

$$\text{Removal Slack Time} = \text{Data Arrival Time} - \text{Data Required Time}$$

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay} + \text{Input Minimum Delay of Pin} + \text{Minimum Pin-to-Register Delay}$$

$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} + \mu t_H$$



If the asynchronous reset signal is from a device pin, you must specify the Input Minimum Delay constraint to the asynchronous reset pin for the Quartus II TimeQuest Timing Analyzer to perform a removal analysis on this path.

Multicycle Paths

Multicycle paths are data paths that require more than one clock cycle to latch data at the destination register. For example, a register may be required to capture data on every second or third rising clock edge. Figure 6–10 shows an example of a multicycle path between a multiplier's input registers and output register where the destination latches data on every other clock edge.

Figure 6–10. Example Diagram of a Multicycle Path

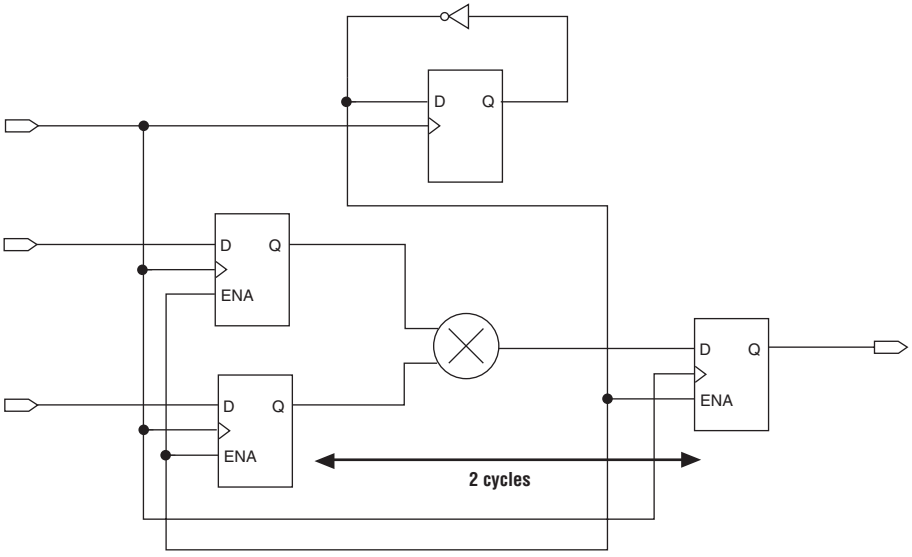


Figure 6–11 shows a register-to-register path where the source clock, `src_clk`, has a period of 10 ns and the destination clock, `dst_clk`, has a period of 5 ns.

Figure 6–11. Register-to-Register Path

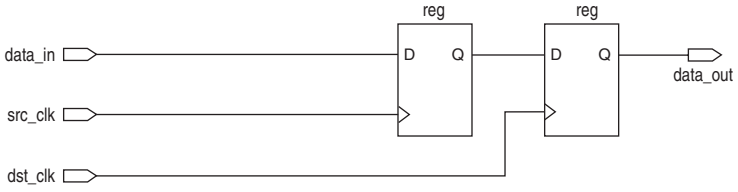
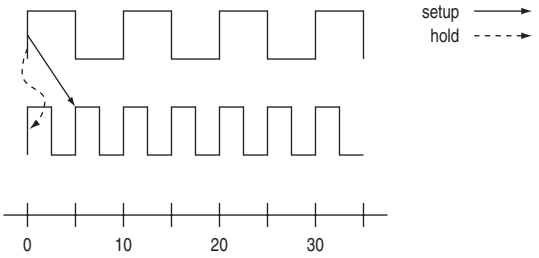


Figure 6–12 shows the respective timing diagrams for the source and destination clocks and the default setup and hold relationships. The default setup relationship is 5 ns and the default hold relationship is 0 ns.

Figure 6–12. Default Setup and Hold Timing Diagram



The default setup and hold relationships can be modified with the `set_multicycle_path` command to accommodate the system requirements.

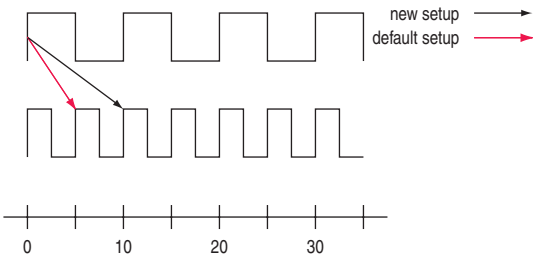
Table 6–3 shows the commands used to modify either the launch or latch edge times that the Quartus II TimeQuest Timing Analyzer Timing Analyzer uses to determine a setup relationship or hold relationship.

Table 6–3. Commands to Modify Edge Times

Command	Description of Modification
`set_multicycle_path -setup -end`	Latch edge time of the setup relationship
`set_multicycle_path -setup -start`	Launch edge time of the setup relationship
`set_multicycle_path -hold -end`	Latch edge time of the hold relationship
`set_multicycle_path -hold -start`	Latch edge time of the hold relationship

Figure 6–13 shows the timing diagram after a multicycle setup of two has been applied. The command moves the latch edge time to 10 ns from the default 5 ns.

Figure 6–13. Modified Setup Diagram



Specify Design Timing Requirements

Next, specify timing constraints and exceptions for your design.

Timing requirements guide the Fitter as it places and routes your design. You must enter all timing constraints and exceptions in an SDC file. This file must be included as part of the project. To add this file to your project, on the Project menu, click **Add/Remove Files in Project**, and add the SDC file in the **Files** dialog box.



Refer to "Specify Timing Constraints" on page 6–20 for a list of timing constraints and exceptions.

After you create the initial database, follow these steps to create timing requirements:

1. Launch the Quartus II TimeQuest Timing Analyzer.
2. Create a Timing Netlist.
3. Specify Timing Constraints.
4. Generate SDC Constraint Reports.



You cannot use the Assignment Editor to specify timing requirements for the TimeQuest timing analyzer.

Create a Timing Netlist

The TimeQuest timing analyzer requires a timing netlist before you can specify timing requirements. The TimeQuest timing analyzer creates a timing netlist based upon the netlist generated in the compilation step. It annotates the timing netlist with either the post-map or post-fit delay results.

You can create a timing netlist in the following ways:

- In the **Tasks** pane, double-click **Create Timing Netlist**.

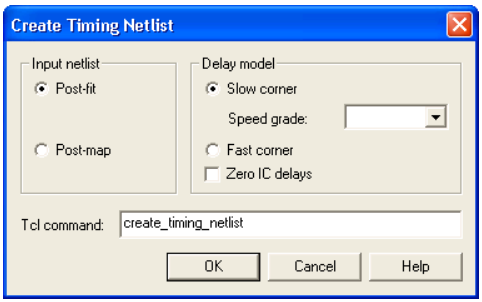
  

  By default, the **Create Timing Netlist** command generates a timing netlist based on the post-fit database. An error message displays if the initial database is a post-map database.

  or

- On the Netlist menu, click **Create Timing Netlist**. The **Create Timing Netlist** dialog box appears (Figure 6–14).

Figure 6–14. Create Timing Netlist Dialog Box



In the **Create Timing Netlist** dialog box, specify the input netlist type and the delay model, and click **OK**.

or

- To create a timing netlist in the **Console** pane, type the following command at a Tcl prompt:

```
create_timing_netlist ↵
```

You can use the `-post_map` option to specify that the timing netlist is based on a post-map database. For example, you can type the following command:

```
create_timing_netlist -post_map ↵
```

Specify Timing Constraints

Use the SDC Editor to create and edit your timing constraints and exceptions. The following constraints are available:

- Create Clock
- Create Generated Clock
- Set Clock Latency
- Set Clock Uncertainty
- Set Input Delay
- Set Output Delay
- Set False Path
- Set Multicycle Path
- Set Maximum Delay
- Set Minimum Delay

For more information on the SDC Editor, refer to "SDC Editor" on page 6–94.

The specified constraints and exceptions guide the Fitter as it places and routes your design. After you have specified all timing constraints and exceptions, save the SDC file.

Generate SDC Constraint Reports

You can generate initial timing reports to verify that all constraints and exceptions have been entered. Because the constraints and exceptions have not been processed by the Fitter, do not use this step to verify that your timing requirements have been met.

You should verify the following items before continuing:

- All clocks are constrained
- Any invalid clock-to-clock transfers have been removed
- All paths are constrained

You can double-click the **Report Clocks** command in the **Tasks** pane, or type the `report_clocks` command in the **Console** pane to verify that all clocks have been properly defined and applied to the proper nodes in the design.

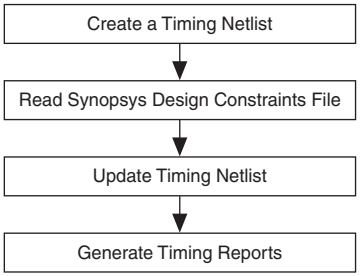
You can double-click the **Report Clock Transfers** command in the **Tasks** pane, or type the `report_clock_transfers` command in the **Console** pane to verify all clock-to-clock transfers.

You can double-click the **Report Unconstrained Paths** command in the **Tasks** pane, or type the `report_ucp` command in the **Console** pane to verify that all paths in the design have been properly constrained.

The Quartus II TimeQuest Timing Analyzer Flow Guidelines

Use the steps shown in Figure 6–15 to verify timing in the TimeQuest timing analyzer.

Figure 6–15. Timing Verification in the TimeQuest Timing Analyzer



The following sections describe each of the steps shown in Figure 6–15.

Create a Timing Netlist

After you perform a full compilation, you must create a timing netlist based on the fully annotated database from the post-fit results.

To create the timing netlist, double-click **Create Timing Netlist** in the **Tasks** pane, or type the following command in the **Console** pane:

```
create_timing_netlist ↵
```

Read the Synopsys Design Constraints File

After you create a timing netlist, you must read an SDC file. This step reads all constraints and exceptions defined in the SDC file.

You can read the SDC file from either the **Task** pane or the **Console** pane.

To read the SDC file from the **Tasks** pane, double-click the **Read SDC File** command.

The **Read SDC File** task reads the <*current revision*>**.sdc** file.

To read the SDC file from the **Console** pane, type the following command in the **Console** pane:

```
read_sdc ↵
```

Update Timing Netlist

You must update the timing netlist after you read an SDC file. The TimeQuest timing analyzer applies all constraints to the netlist for verification, and removes any invalid or false paths in the design from verification.

To update the timing netlist, double-click **Update Timing Netlist** in the **Tasks** pane, or type the following command in the **Console** pane:

```
update_timing_netlist ↵
```

Generate Timing Reports

You can generate timing reports for all critical paths in your design. The **Tasks** pane contains the commonly used reporting commands. Individual or custom reports can be generated for your design.

For more information about reporting, refer to the section "Timing Reports" on page 6–58.



For a full list of available report APIs, refer to the *SDC & TimeQuest API Reference Manual*.

As you verify timing, you may encounter failures along critical paths. If this occurs, you can refine the existing constraints or create new ones to change the effects of existing constraints. If you modify, remove, or add constraints, you should perform a full compilation. This allows the Fitter to re-optimize the design based upon the new constraints, and brings you back to the Perform Compilation step in the process. This iterative process allows you to resolve your timing violations in the design.



For a sample Tcl script to automate the timing analysis flow, refer to the *TimeQuest Quick Start Tutorial*.

Collections

The Quartus II TimeQuest Timing Analyzer supports collection APIs that provide easy access to ports, pins, cells, or nodes in the design. Use collection APIs with any valid constraints or Tcl commands specified in the Quartus II TimeQuest Timing Analyzer.

Table 6–4 describes the collection commands supported by the Quartus II TimeQuest Timing Analyzer.

Table 6–4. Collection Commands

Command	Description
`all_clocks`	Returns a collection of all clocks in the design.
`all_inputs`	Returns a collection of all input ports in the design.
`all_outputs`	Returns a collection of all output ports in the design.
`all_registers`	Returns a collection of all registers in the design.
`get_cells`	Returns a collection of cells in the design. All cell names in the collection match the specified pattern. Wildcards can be used to select multiple cells at the same time.
`get_clocks`	Returns a collection of clocks in the design. When used as an argument to another command, such as the `-from` or `-to` of `set_multicycle_path`, each node in the clock represents all nodes clocked by the clocks in the collection. The default uses the specific node (even if it is a clock) as the target of a command.
`get_nets`	Returns a collection of nets in the design. All net names in the collection match the specified pattern. You can use wildcards to select multiple nets at the same time.
`get_pins`	Returns a collection of pins in the design. All pin names in the collection match the specified pattern. You can use wildcards to select multiple pins at the same time.
`get_ports`	Returns a collection of ports (design inputs and outputs) in the design.

Table 6–5 describes the SDC extension collection commands supported by the Quartus II TimeQuest Timing Analyzer.

Table 6–5. SDC Extension Collection Commands (Part 1 of 2)

Command	Description
`get_fanouts` <*filter*>	Returns a collection of fan-out nodes starting from <*filter*>.
`get_keepers` <*filter*>	Returns a collection of keeper nodes (non-combinational nodes) in the design.
`get_nodes` <*filter*>	Returns a collection of nodes in the design. The get_nodes collection cannot be used when specifying constraints or exceptions.
`get_partitions` <*filter*>	Returns a collection of partitions matching the <*filter*>.
`get_registers` <*filter*>	Returns a collection of registers in the design.
`get_fanins` <*filter*>	Returns a collection of fan-in nodes starting from <*filter*>.
`derive_pll_clocks`	Automatically create generated clocks on the outputs of PLL. The generated clock properties reflect the PLL properties that have been specified by the MegaWizard® Plug-In Manager.

Table 6–5. SDC Extension Collection Commands (Part 2 of 2)

Command	Description
`get_assignment_groups` <*filter*>	Returns either a collection of keepers, ports, or registers that have been saved to the Quartus Settings File (QSF) with the Assignment (Time) Groups option.
`remove_clock` <*clock list*>	Removes the list of clocks specified by <*clock list*>.
`set_scc_mode` <*size*>	Allows you to set the maximum Strongly Connected Components (SCC) loop size or force the Quartus II TimeQuest Timing Analyzer to always estimate delays through SCCs.
`set_time_format`	Sets time format, including time unit and decimal places.



For more information about collections, refer to the SDC file and the *SDC and TimeQuest API Reference Manual*.

Application Examples

Example 6–1 shows various uses of the `create_clock` and `create_generated_clock` commands and specific design structures.

Example 6–1. create_clock and create_generate_clock Commands and Specific Design Structures

```
# Create a simple 10 ns with clock with a 60 % duty cycle
create_clock -period 10 -waveform {0 6} -name clk [get_ports clk]
# The following multicycle applies to all paths ending at registers
# clocked by clk
set_multicycle_path -to [get_clocks clk] 2
```

Constraints Files

The Quartus II TimeQuest Timing Analyzer stores all timing constraints in an SDC file. You can create an SDC file with different constraints for place-and-route and for timing analysis.



The SDC file should contain only SDC and Tcl commands. Commands to manipulate the timing netlist or control the compilation flow should not be included in the SDC file.

The Quartus II software does not automatically update SDC files. You must explicitly write new or updated constraints in the TimeQuest GUI. Use the `write_sdc` command, or, in the Quartus II TimeQuest Timing Analyzer, on the Constraints menu, click **Write SDC File** to write your constraints to an SDC file.

Fitter and Timing Analysis SDC Files

You can specify the same or different SDC files for the Quartus II Fitter for place-and-route, and the Quartus II TimeQuest Timing Analyzer for static timing analysis. Using different SDC files allows you to have one set of constraints for place-and-route, and another set of constraints for final timing sign-off in the Quartus II TimeQuest Timing Analyzer.

Specifying SDC Files for Place-and-Route

To specify an SDC file for the Fitter, you must add the SDC file to your Quartus II project. To add the file to your project, use the following command in the Tcl console:

```
set_global_assignment -name SDC_FILE <SDC file name>
```

Or, in the Quartus II GUI, on the Project menu, click **Add/Remove Files in Project**.

The Fitter optimizes your design based on the requirements in the SDC files in your project.

The results shown in the timing analysis report located in the Compilation Report are based on the SDC files added to the project.



You must specify the Quartus II TimeQuest Timing Analyzer as the default timing analyzer to make the Fitter read the SDC file.

Specifying SDC Files for Static Timing Analysis

After you create a timing netlist in the Quartus II TimeQuest Timing Analyzer, you must specify timing constraints and exceptions before you can perform a timing analysis. The timing requirements do not have to be identical to those provided to the Fitter. You can specify your timing requirements manually or you can read a previously created SDC file.

To manually enter your timing requirements, you can use constraint entry dialog boxes or SDC commands. If you have an SDC file that contains your timing requirements, you can use this file to apply your timing requirements. To specify the SDC file for timing analysis in the Quartus II TimeQuest Timing Analyzer, use the following command:

```
read_sdc [<SDC file name>]
```

If you use the TimeQuest GUI to apply SDC file for timing analysis, in the Quartus II TimeQuest Timing Analyzer, on the Constraints menu, click **Read SDC File**.



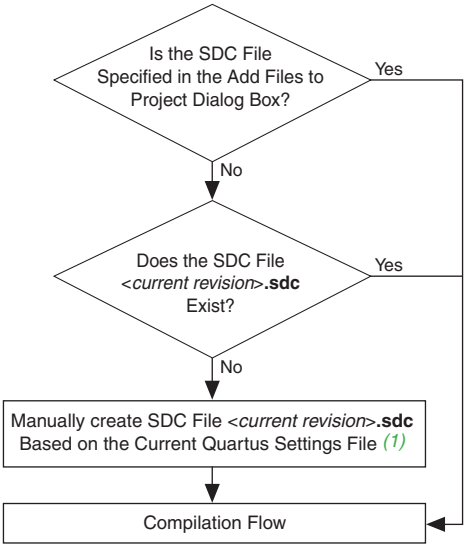
By default, the **Read SDC File** command in the **Tasks** pane reads the SDC files specified in the Quartus II Settings File (**.qsf**) (QSF), which are the same SDC files used by the Fitter.

Synopsys Design Constraints File Precedence

The Quartus II Fitter and the Quartus II TimeQuest Timing Analyzer reads the SDC files from the files list in the QSF file in the order they are listed, from top to bottom.

The Quartus II software searches for an SDC file, as shown in Figure 6–16.

Figure 6–16. Synopsys Design Constraints File Order of Precedence



Note to Figure 6–16:

(1) This occurs only in the Quartus II TimeQuest Timing Analyzer, and not during compilation in the Quartus II software. The Quartus II TimeQuest Timing Analyzer has the ability to automate the conversion of the QSF timing assignments to SDC if no SDC file exists when the Quartus II TimeQuest Timing Analyzer is opened.



If you type the `read_sdc` command at the command line without any arguments, the precedence order shown in Figure 6–16 is followed.

Clock Specification

The specification of all clocks and any associated clock characteristics in your design is essential for accurate static timing analysis results. The Quartus II TimeQuest Timing Analyzer supports many SDC commands that accommodate various clocking schemes and any clock characteristics.

This section describes the SDC file API available to create and specify clock characteristics.

Clocks

Use the `create_clock` command to create a clock at any register, port, or pin. You can create each clock with unique characteristics. Example 6–2 shows the `create_clock` command and options.

Example 6–2. create_clock Command

```
create_clock
-period <period value>
[-name <clock name>]
[-waveform <edge list>]
[-add]
<targets>
```

Table 6–6 describes the options for the `create_clock` command.

Table 6–6. create_clock Command Options

Option	Description
`-period` <*period value*>	Specifies the clock period. You can also specify the clock period in units of frequency, such as `-period` <*num*>`MHz`. *(1)*
`-name` <*clock name*>	Name of the specific clock, for example, `sysclock`. If you do not specify the clock name, the clock name is the same as the node to which it is assigned.
`-waveform` <*edge list*>	Specifies the clock's rising and falling edges. The edge list alternates between rising edge and falling edge. For example, a 10 ns period where the first rising edge occurs at 0 ns and the first falling edge occurs at 5 ns would be written as `-waveform {0 5}`. The difference must be within one period unit, and the rise edge must come before the fall edge. The default edge list is `{0` <*period*>`/2}`, or a 50% duty cycle.
`-add`	Allows you to specify more than one clock to the same port or pin.
<*targets*>	Specifies the port(s) or pin(s) to which the assignment applies. If source objects are not specified, the clock is a virtual clock. Refer to "Virtual Clocks" on page 6–32 for more information.

Note to Table 6–6:

(1) The default time unit in the Quartus II TimeQuest Timing Analyzer is nanoseconds (ns).

Example 6–3 shows how to create a 10 ns clock with a 50% duty cycle, where the first rising edge occurs at 0 ns applied to port clk.

Example 6–3. 100MHz Clock Creation

```
create_clock -period 10 -waveform { 0 5 } clk
```

Example 6–4 shows how to create a 10 ns clock with a 50% duty cycle that is phase shifted by 90 degrees applied to port clk_sys.

Example 6–4. 100MHz Shifted by 90 Degrees Clock Creation

```
create_clock -period 10 -waveform { 2.5 7.5 } clk_sys
```

Clocks defined with the create_clock command have a default source latency value of zero. The Quartus II TimeQuest Timing Analyzer automatically computes the clock's network latency for non-virtual clocks.

Generated Clocks

The Quartus II TimeQuest Timing Analyzer considers clock dividers, ripple clocks, or circuits that modify or change the characteristics of the incoming or master clock as generated clocks. You should define the output of these circuits as generated clocks. This definition allows the Quartus II TimeQuest Timing Analyzer to analyze these clocks and account for any network latency associated with them.

Use the create_generated_clock command to create generated clocks. Example 6–5 shows the create_generated_clock command and the available options.

Example 6–5. create_generated_clock Command

```
create_generated_clock
[-name <clock name>]
-source <master pin>
[-edges <edge list>]
[-edge_shift <shift list>]
[-divide_by <factor>]
[-multiply_by <factor>]
[-duty_cycle <percent>]
[-add]
[-invert]
[-master_clock <clock>]
[-phase <phase>]
[-offset <offset>]
<targets>
```

Table 6–7 describes the options for the `create_generated_clock` command.

Table 6–7. create_generated_clock Command Options

Option	Description
`-name` <*clock name*>	Name of the generated clock, for example, `clk_x2`. If you do not specify the clock name, the clock name is the same as the first node to which it is assigned.
`-source` <*master pin*>	The <*master pin*> specifies the node in the design from which the clock settings derive.
`-edges` <*edge list*> | `-edge_shift` <*shift list*>	The `-edges` option specifies the new rising and falling edges with respect to the master clock's rising and falling edges. The master clock's rising and falling edges are numbered `1..`<*n*> starting with the first rising edge, for example, edge 1. The first falling edge after that is edge number 2, the next rising edge number 3, and so on. The <*edge list*> must be in ascending order. The same edge may be used for two entries to indicate a clock pulse independent of the original waveform's duty cycle. `-edge_shift` specifies the amount of shift for each edge in the <*edge list*>. The `-invert` option can be used to invert the clock after the `-edges` and `-edge_shifts` are applied. *(1)*
`-divide_by` <*factor*> | `-multiply_by` <*factor*>	The `-divide_by` and `-multiply_by` factors are based on the first rising edge of the clock, and extend or contract the waveform by the specified factors. For example, a `-divide_by 2` is equivalent to `-edges {1 3 5}`. For multiplied clocks, the duty cycle can also be specified. The Quartus II TimeQuest Timing Analyzer supports specifying multiply and divide factors at the same time.
`-duty_cycle` <*percent*>	Specifies the duty cycle of the generated clock. The duty cycle is applied last.
`-add`	Allows you to specify more than one clock to the same pin.
`-invert`	Inversion is applied at the output of the clock after all other modifications are applied, except duty cycle.
`-master_clock` <*clock*>	`-master_clock` is used to specify the clock if multiple clocks exist at the master pin.
`-phase` <*phase*>	Specifies the phase of the generated clock.
`-offset` <*offset*>	Specifies the offset of the generated clock.
<*targets*>	Specifies the port(s) or pin(s) to which the assignment applies.

Note to Table 6–7:
(1) The Quartus II TimeQuest Timing Analyzer supports a maximum of three edges in the edge list.

Source latencies are based on clock network delays from the master clock (not necessarily the master pin). You can use the `set_clock_latency -source` command to override the source latency.

Figure 6–17 shows how to create an inverted generated clock based on a 10 ns clock.

Figure 6–17. Generating an Inverted Clock

```
create_clock -period 10 [get_ports clk]
create_generated_clock -divide_by 1 -invert -source [get_registers clk] \
    [get_registers gen|clkreg]
```

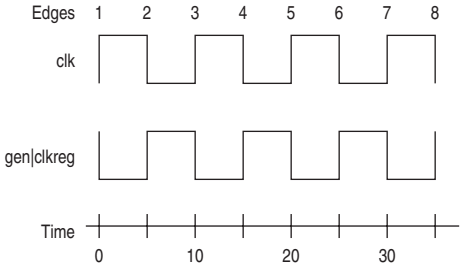


Figure 6–18 shows how to modify the generated clock using the `-edges` and `-edge_shift` options.

Figure 6–18. Edges and Edge Shifting a Generated Clock

```
create_clock -period 10 -waveform { 0 5} [get_ports clk]
# Creates a divide-by-t clock
create_generated_clock -source [get_ports clk] -edges {1 3 5 } [get_registers \
clkdivA|clkreg]
# Creates a divide-by-2 clock independent of the master clocks' duty cycle (now 50%)
create_generated_clock -source [get_ports clk] -edges {1 1 5} -edge_shift { 0 2.5 0 } \
[get_registers clkdivB|clkreg]
```

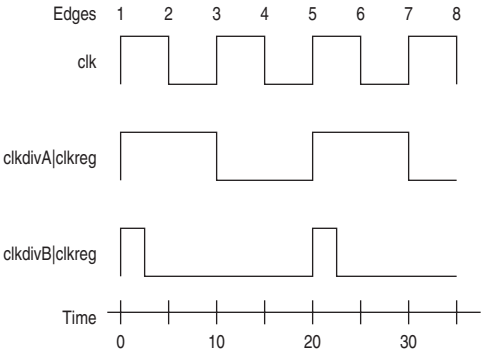
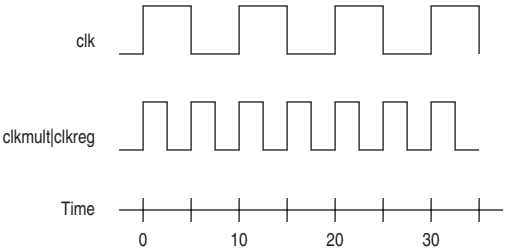


Figure 6–19 shows the effect of the `-multiply_by` option on the generated clock.

Figure 6–19. Multiplying a Generated Clock

```
create_clock -period 10 -waveform { 0 5 } [get_ports clk]
# Creates a multiply-by-2 clock
create_generated_clock -source [get_ports clk] -multiply_by 2 [get_registers \
clkmult|clkreg]
```



Virtual Clocks

A virtual clock is a clock that does not have a real source in the design or that does not interact directly with the design. For example, if a clock feeds only an external device's clock port, and not a clock port in the design, and the external device then feeds (or is fed by) a port in the design, it is considered a virtual clock.

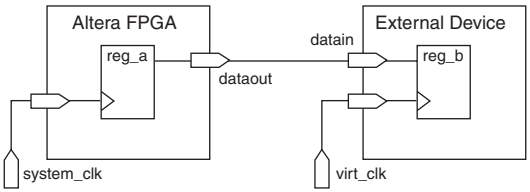
Use the `create_clock` command to create virtual clocks, with no value specified for the `source` option.



You can use virtual clocks for `set_input_delay` and `set_output_delay` constraints.

Figure 6–20 shows an example where a virtual clock is required for the Quartus II TimeQuest Timing Analyzer to properly analyze the relationship between the external register and those in the design. Because the oscillator labeled `virt_clk` does not interact with the Altera device, but acts as the clock source for the external register, the clock `virt_clk` must be declared. Example 6–6 shows the command to create a 10 ns virtual clock named `virt_clk` with a 50% duty cycle where the first rising edge occurs at 0 ns. The virtual clock is then used as the clock source for an output delay constraint.

Figure 6–20. Virtual Clock Board Topology



After you create the virtual clock, you can perform register-to-register analysis between the register in the Altera device and the register in the external device.

Example 6–6. Virtual Clock Example 1

```
#create base clock for the design
create_clock -period 5 [get_ports system_clk]
#create the virtual clock for the external register
create_clock -period 10 -name virt_clk -waveform { 0 5 }
#set the output delay referencing the virtual clock
set_output_delay -clock virt_clk -max 1.5 [get_ports dataout]
```

Example 6–7 shows the command to create a 10 ns virtual clock with a 50% duty cycle that is phase shifted by 90 degrees.

Example 6–7. Virtual Clock Example 2

```
create_clock -name virt_clk -period 10 -waveform { 2.5 7.5 }
```

Multi-Frequency Clocks

Certain designs have more than one clock source feeding a single clock port in the design. The additional clock may act as a low power clock, with a lower frequency than the primary clock. To analyze this type of design, the `create_clock` command supports the `-add` option that allows you to add more than one clock to a clock node.

Example 6–8 shows the command to create a 10 ns clock applied to clock port `clk`, and then add an additional 15 ns clock to the same clock port. The Quartus II TimeQuest Timing Analyzer uses both clocks when it performs timing analysis.

Example 6–8. Multi-Frequency Example

```
create_clock -period 10 -name clock_primary -waveform { 0 5 } [get_ports clk]
create_clock -period 15 -name clock_secondary -waveform { 0 7.5 } [get_ports clk] -add
```

Automatic Clock Detection

To create clocks for all clock nodes in your design automatically, use the `derive_clocks` command. This command creates clocks on ports or registers to ensure every register in the design has a clock.

Example 6–9 shows the `derive_clocks` command and options.

Example 6–9. derive_clocks Command

```
derive_clocks
[-period <period value>]
[-waveform <edge list>]
```

Table 6–8 describes the options for the `derive_clocks` command.

Table 6–8. derive_clocks Command Options

Option	Description
`-period` <*period value*>	Creates the clock period. You can also specify the frequency as `-period` <*num*>`MHz`. *(1)*
`-waveform` <*edge list*>	Creates the clock's rising and falling edges. The edge list alternates between rising edge and falling edge. For example, for a 10 ns period where the first rising edge occurs at 0 ns and first falling edge occurs at 5 ns, the edge list is `waveform {0 5}`. The difference must be within one period unit, and the rising edge must come before the falling edge. The default edge list is `{0 period/2}`, or a 50% duty cycle.

Note to Table 6–8:
(1) This option uses the default time unit nanoseconds (ns).



The `derive_clocks` command does not create clocks for the outputs of the PLLs.

The `derive_clocks` command is equivalent to using `create_clock` for each register or port feeding the clock pin of a register.



The use of the `derive_clocks` command for final timing sign-off is not recommended. You should create clocks for all clock sources using the `create_clock` and `create_generated_clock` commands.

Derive PLL Clocks

PLLs are used for clock management and synthesis in Altera devices. You can customize the clocks generated from the outputs of the PLL based on the design requirements. Because a clock should be created for all clock nodes, all outputs of the PLL should have an associated clock.

You can manually create a clock for each output of the PLL with the `create_generated_clock` command, or you can use the `derive_pll_clocks` command, which automatically searches the timing netlist and creates generated clocks for all PLL outputs according to the settings specified for each PLL output.

Use the `derive_pll_clocks` command to automatically create a clock for each output of the PLL, as shown in the following list:

```
derive_pll_clocks
[-use_tan_name]
```

Table 6–9 describes the options for the `derive_pll_clocks` command.

Table 6–9. derive_pll_clocks Command Options

Option	Description
`-use_tan_name`	By default, the clock name is the output clock name. This option uses the net name similar to the names used by the Quartus II Classic Timing Analyzer.

The `derive_pll_clocks` command calls the `create_generated_clock` command to create generated clocks on the outputs of the PLL. The source for the `create_generated_clock` command is the input clock pin of the PLL. Before or after the `derive_pll_clocks` command has been issued, you must manually create a base clock for the input clock port of the PLL. If a clock is not defined for the input clock node of the PLL, no clocks are reported for the PLL outputs. Instead, the Quartus II TimeQuest Timing Analyzer issues a warning message similar to Figure 6–10 when the timing netlist is updated.

Example 6–10. Warning Message

```
Warning: The master clock for this clock assignment could not be derived.
Clock: <name of PLL output clock pin name> was not created.
```

You can include the `derive_pll_clocks` command in your SDC file, which allows the `derive_pll_clocks` command to automatically detect any changes to the PLL. With the `derive_pll_clocks`

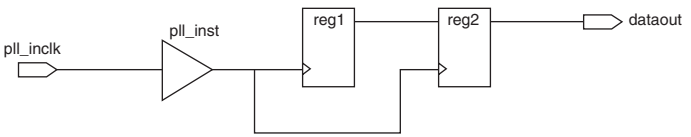
command in your SDC file, each time the file is read, the appropriate `create_generated_clock` command for the PLL output clock pin is generated. If you use the `write_sdc` command after the `derive_pll_clock` command, the new SDC file contains the individual `create_generated_clock` commands for the PLL output clock pins and not the `derive_pll_clocks` command. Any changes to the properties of the PLL are not automatically reflected in the new SDC file. You must manually update the `create_generated_clock` commands in the new SDC file written by the `derive_pll_clocks` command, to reflect the changes to the PLL.



The `derive_pll_clocks` constraint will also constrain any LVDS transmitters or LVDS receivers in the design by adding the appropriate multicycle constraints to account for any deserialization factors.

For example, Figure 6–21 shows a simple PLL design with a register-to-register path.

Figure 6–21. Simple PLL Design



Use the `derive_pll_clocks` command to automatically constrain the PLL. When this command is issued for the design shown in Figure 6–21, the messages shown in Example 6–11 are generated.

Example 6–11. derive_pll_clocks Generated Messages

```
Info:
Info: Deriving PLL Clocks:
Info: create_generated_clock -source pll_inst|altpll_component|pll|inclk[0] -divide_by 2 -name
pll_inst|altpll_component|pll|CLK[0] pll_inst|altpll_component|pll|clk[0]
Info:
```

The node name `pll_inst|altpll_component|pll|inclk[0]` used for the source option refers to the input clock pin of the PLL. In addition, the name of the output clock of the PLL is the name of the PLL output clock node, `pll_inst|altpll_component|pll|clk[0]`.



If the PLL is in clock switchover mode, multiple clocks are created for the output clock of the PLL; one for the primary input clock (for example, `inclk[0]`), and one for the secondary input clock (for example, `inclk[1]`). In this case, you should cut the primary and secondary output clocks using the `set_clock_groups` command with the `-exclusive` option.

Before you can generate any reports for this design, you must create a base clock for the PLL input clock port. Use a command similar to the following:

```
create_clock -period 5 [get_ports pll_inclk]
```



You do not have to generate the base clock on the input clock pin of the PLL: `pll_inst|altpll_component|pll|inclk[0]`. The clock created on the PLL input clock port propagates to all fan-outs of the clock port, including the PLL input clock pin.

Default Clock Constraints

To provide a complete clock analysis, the Quartus II TimeQuest Timing Analyzer, by default, automatically creates clocks for all detected clock nodes in your design that have not be constrained, if there are no base clock constraints in the design. The Quartus II TimeQuest Timing Analyzer creates a base clock with a 1 GHz requirement to unconstrained clock nodes, using the following command:

```
derive_clocks -period 1
```



Individual clock constraints (for example, `create_clock`, `create_generated_clock`) should be made for all clocks in the design. This results in a thorough and realistic analysis of the design's timing requirements. The use of `derive_clocks` should be avoided for final timing sign-off.

The default clock constraint is only applied when the Quartus II TimeQuest Timing Analyzer detects that all synchronous elements have no clocks associated with them. For example, if a design contains two clocks and only one clock has constraints, the default clock constraint will not be applied. However, if both clocks have not been constrained, the default clock constraint will be applied.

Clock Groups

Many clocks can exist in a design, however, not all of the clocks interact with one another, and certain clock interactions are not possible. Asynchronous clocks are unrelated clocks (asynchronous clocks have

different ideal clock sources). Exclusive clocks are not active at the same time (for example, multiplexed clocks). The mutual exclusivity must be declared to the Quartus II TimeQuest Timing Analyzer to prevent it from analyzing these clock interactions.

Use the `set_clock_groups` command to specify clocks that are exclusive or asynchronous. Example 6–12 shows the `set_clock_groups` command and options.

Example 6–12. set_clock_groups Command

```
set_clock_groups
[-asynchronous | -exclusive]
-group <clock name>
[-group <clock name>]
[-group <clock name>] ...
```

Table 6–10 describes the options for the `set_clock_groups` command.

Table 6–10. set_clock_groups Command Options

Option	Description
`-asynchronous`	Asynchronous clocks—when the two clocks have no phase relationship and are active at the same time.
`-exclusive`	Exclusive clocks—when only one of the two clocks will be active at any given time. An example of an exclusive clock group is when two clocks feed a 2-to-1 MUX.
`-group` <*clock name*>	Specifies valid destination clock names that are mutually exclusive. <*clock name*> is used to specify the clock names.

Example 6–13 shows a `set_clock_groups` command and the equivalent `set_false_path` commands.

Example 6–13. set_clock_groups Example

```
# Clocks A and C are never active when clocks B and D are active
set_clock_groups -exclusive -group {A C} -group {B D}

# Equivalent specification using false paths
set_false_path -from [get_clocks A] -to [get_clocks B]
set_false_path -from [get_clocks A] -to [get_clocks D]
set_false_path -from [get_clocks C] -to [get_clocks B]
set_false_path -from [get_clocks C] -to [get_clocks D]
set_false_path -from [get_clocks B] -to [get_clocks A]
set_false_path -from [get_clocks B] -to [get_clocks C]
set_false_path -from [get_clocks D] -to [get_clocks A]
set_false_path -from [get_clocks D] -to [get_clocks C]
```

Clock Effect Characteristics

The `create_clock` and `create_generated_clock` commands create ideal clocks that do not account for any board effects. This section describes how to account for clock effect characteristics with clock latency and clock uncertainty.

Clock Latency

There are two forms of clock latency: source and network. Source latency is the propagation delay from the origin of the clock to the clock definition point (for example, a clock port), and network latency is the propagation delay from a clock definition point to a register's clock pin. The total latency (or clock propagation delay) at a register's clock pin is the sum of the source and network latencies in the clock path.



The `set_clock_latency` command supports only source latency. The `-source` option must be specified when using the command.

Use the `set_clock_latency` command to specify source latency to any clock ports in the design. Example 6–14 shows the `set_clock_latency` command and options.

Example 6–14. set_clock_latency Command

```
set_clock_latency
-source
[-clock <clock_list>]
[-rise | -fall]
[-late | -early]
<delay>
<targets>
```

Table 6–11 describes the options for the `set_clock_latency` command.

Table 6–11. set_clock_latency Command Options (Part 1 of 2)

Option	Description
`-source`	Specifies a source latency.
`-clock <clock list>`	Specifies the clock to use if the target has more than one clock assigned to it.

Table 6–11. set_clock_latency Command Options (Part 2 of 2)

Option	Description
`-rise	-fall`	Specifies the rising or falling delays.
`-late	-early`	Specifies the earliest or the latest arrival times to the clock.
<*delay*>	Specifies the delay value.
<*targets*>	Specifies the clocks or clock sources if a clock is clocked by more than one clock.

The Quartus II TimeQuest Timing Analyzer automatically computes network latencies; therefore, the `set_clock_latency` command specifies only the source latencies.

Clock Uncertainty

The `set_clock_uncertainty` command specifies clock uncertainty or skew for clocks or clock-to-clock transfers. Specify the uncertainty separately for setup and hold, and you can specify separate rising and falling clock transitions. The Quartus II TimeQuest Timing Analyzer subtracts the setup uncertainty from the data required time for each applicable path, and adds the hold uncertainty to the data required time for each applicable path.

Use the `set_clock_uncertainty` command to specify any clock uncertainty to the clock port. Example 6–15 shows the `set_clock_uncertainty` command and options.

Example 6–15. set_clock_uncertainty Command and Options

```
set_clock_uncertainty
[-rise_from <rise from clock> | -fall_from <fall from clock> |
-from <from clock>]
[-rise_to <rise to clock> | -fall_to <fall to clock> | -to <to clock>]
[-setup | -hold]
<value>
```

Table 6–12 describes the options for the set_clock_ uncertainty command.

Table 6–12. Options Description for set_clock_uncertainty

Option	Description
-from <*from clock*>	Specifies the from clock.
-rise_from <*rise from clock*>	Specifies the rise from clock.
-fall_from <*fall from clock*>	Specifies the fall from clock.
-to <*to clock*>	Specifies the to clock.
-rise_to <*rise to clock*>	Specifies the rise to clock.
-fall_to <*fall to clock*>	Specifies the fall to clock.
-setup | -hold	Specifies setup or hold.
<*value*>	Uncertainty value.

Derive Clock Uncertainty

Use the derive_clock_uncertainty command to automatically apply inter-clock, intra-clock, and I/O interface uncertainties. Both setup and hold uncertainties are calculated for each clock-to-clock transfer. Example 6–16 shows the derive_clock_uncertainty command and options.

Example 6–16. derive_clock_uncertainty Command

```
derive_clock_uncertainty
[-overwrite]
[-dtw]
```

Table 6–13 describes the options for the derive_clock_uncertainty command.

Table 6–13. derive_clock_uncertainty Command Options

Option	Description
-overwrite	Overwrites previously performed clock uncertainty assignments
-dtw	Creates the **PLLJ-PLLSPE_INFO.txt** file

The Quartus II TimeQuest Timing Analyzer automatically applies clock uncertainties to clock-to-clock transfers in the design.

Any clock uncertainty constraints that have been applied to source and destination clock pairs with the `set_clock_uncertainty` command have a higher precedence than the clock uncertainties derived from the `derive_clock_uncertainty` command for the same source and destination clock pairs. For example, if the `set_clock_uncertainty` command is called first to specify clock uncertainties between the source clock `CLKA` and destination clock `CLKB`. Then the `derive_clock_uncertainty` command is called second, the clock uncertainty calculated by the `derive_clock_uncertainty` command is ignored for the source clock `CLKA` to destination clock `CLKB`.

The clock uncertainty value that would have been used, however, is still reported for informational purposes. You can use the `-overwrite` command to overwrite previous clock uncertainty assignments, or remove them manually with the `remove_clock_uncertainty` command.

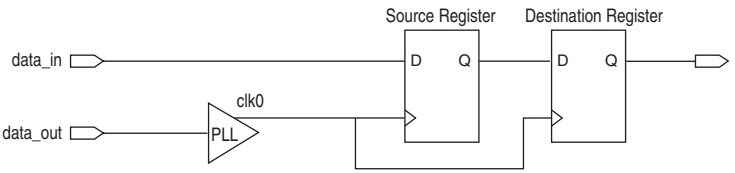
In the following types of clock-to-clock transfers, clock certainties can arise. They are modeled by the `derive_clock_uncertainty` command automatically.

- Inter-clock
- Intra-clock
- I/O Interface

Inter-Clock Transfers

Inter-clock transfers occur when the register-to-register transfer happens in the core of the FPGA and source and destination clocks come from the same PLL output pin or clock port. An example of an inter-clock transfer is shown in Figure 6–22.

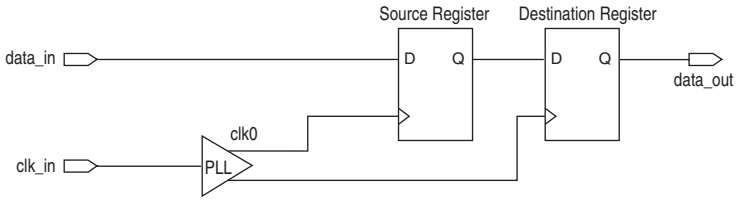
Figure 6–22. Inter-Clock Transfer



Intra-Clock Transfers

Intra-clock transfers occur when the register-to-register transfer happens in the core of the FPGA and source and destination clocks come from a different PLL output pin or clock port. An example of an intra-clock transfer is shown in Figure 6–23.

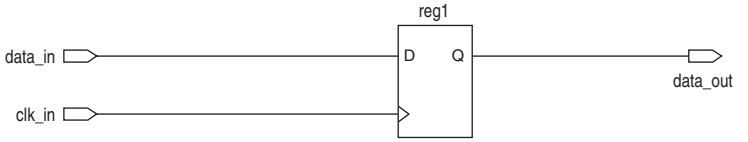
Figure 6–23. Intra-Clock Transfer



I/O Interface Clock Transfers

I/O interface clock transfers occur when data transfers from an I/O port to the core of the FPGA (input) or from the core of the FPGA to the I/O port (output). An example of an I/O interface clock transfer is shown in Figure 6–24.

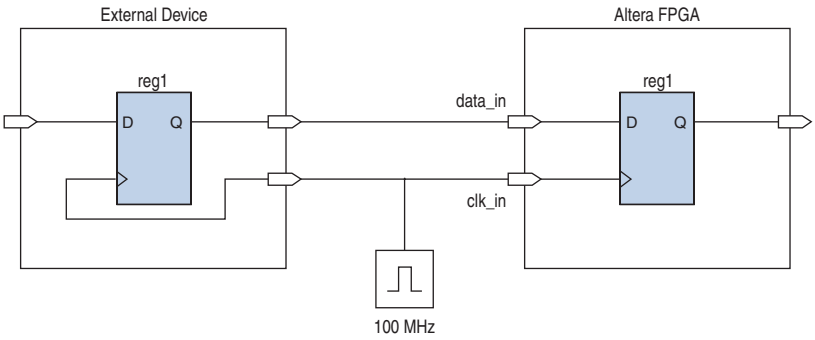
Figure 6–24. Interface-Clock Transfer



For I/O interface uncertainty, you must create a virtual clock and constrain the input and output ports with the `set_input_delay` and `set_output_delay` commands that reference the virtual clock. The virtual clock is required to prevent the `derive_clock_uncertainty` command from applying clock certainties for either intra- or inter-clock transfers on an I/O interface clock transfer when the `set_input_delay` or `set_output_delay` commands reference a clock port or PLL output. If a virtual clock is not referenced in the `set_input_delay` or `set_output_delay` commands to clock uncertainty for the I/O interface clock, transfers will result in a pessimistic analysis.

The virtual clock should be created with the same properties as the original clock that is driving the I/O port. For example, Figure 6–25 shows a typical input I/O interface with the clock specifications.

Figure 6–25. I/O Interface Specifications



Example 6–17 shows the SDC commands to constrain the I/O Interface shown in Figure 6–25.

Example 6–17. SDC Commands to Constrain the I/O Interface

```
# Create the base clock for the clock port
create_clock -period 10 -name clk_in [get_ports clk_in]
# Create a virtual clock with the same properties of the base clock driving
# the source register
create_clock -period 10 -name virt_clk_in
# Create the input delay referencing the virtual clock and not the base
# clock
# DO NOT use set_input_delay -clock clk_in <delay_value>
# [get_ports data_in]
set_input_delay -clock virt_clk_in <delay value> [get_ports data_in]
```

I/O Specifications

The Quartus II TimeQuest Timing Analyzer supports Synopsys Design Constraints that constrain the ports in your design. These constraints allow the Quartus II TimeQuest Timing Analyzer to perform a system static timing analysis that includes not only the FPGA timing, but also any external device timing and board timing parameters.

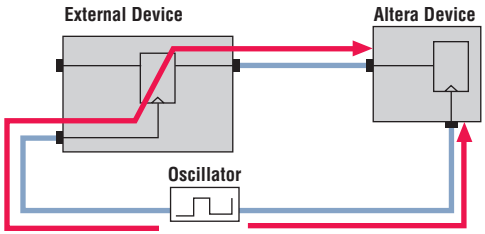
Input and Output Delay

Use input and output delay constraints to specify any external device or board timing parameters. When you apply these constraints, the Quartus II TimeQuest Timing Analyzer performs static timing analysis on the entire system.

Set Input Delay

The `set_input_delay` constraint specifies the data arrival time at a port (a device I/O) with respect to a given clock. Figure 6–26 shows an input delay path.

Figure 6–26. Set Input Delay



Use the `set_input_delay` command to specify input delay constraints to ports in the design. Example 6–18 shows the `set_input_delay` command and options.

Example 6–18. set_input_delay Command

```
set_input_delay
-clock <clock name>
[-clock_fall]
[-rise | -fall]
[-max | -min]
[-add_delay]
[-reference_pin <target>]
[-source_latency_included]
<delay value>
<targets>
```

Table 6–14 describes the options for the `set_input_delay` command.

Table 6–14. set_input_delay Command Options

Option	Description
`-clock` <*clock name*>	Specifies the source clock.
`-clock_fall`	Specifies the arrival time with respect to the falling edge of the clock.
`-rise | -fall`	Specifies either the rise or fall delay at the port.
`-max | -min`	Specifies the minimum or maximum data arrival time.
`-add_delay`	Adds another delay, but does not replace the existing delays assigned to the port.
`-reference_pin` <*target*>	Specifies a pin or port in the design from which to determine source and network latencies. This is useful to specify input delays relative to an output port fed by a clock.
`-source_latency_ included`	Specifies that the input delay value includes the source latency delay value, and therefore any source clock latency assigned to the clock will be ignored.
<*delay value*>	Specifies the delay value.
<*targets*>	Specifies the destination ports or pins.



A warning message appears if you specify only a `–max` or `–min` value for the input delay value. The input minimum delay default value is the input maximum delay, and the input maximum delay default value is the input minimum delay, if only one is specified. Similarly, a warning message appears if you specify only a `-rise` or `-fall` value for the delay value, and the delay defaults in the same manner as the input minimum and input maximum delays.

The maximum value is used for setup checks, and the minimum value is used for hold checks.

By default, a set of input delays (min/max, rise/fall) is allowed for only one clock, `-clock_fall`, `-reference_pin` combination. Specifying an input delay on the same port for a different clock, `-clock_fall`, or `-reference_pin` removes any previously set input delays, unless you specify the `-add_delay` option. When you specify the `-add_delay` option, the worst-case values are used.

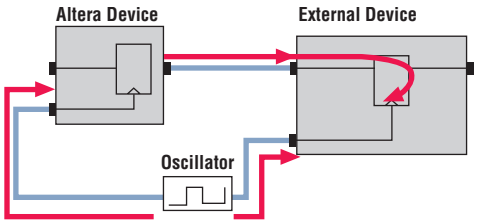
The `-rise` and `-fall` options are mutually exclusive. The `-min` and `-max` options are also mutually exclusive.

Set Output Delay

The `set_output_delay` command specifies the data required time at a port (the device pin) with respect to a given clock.

Use the `set_output_delay` command to specify output delay constraints to ports in the design. Figure 6–27 shows an output delay path.

Figure 6–27. Output Delay



Example 6–19 shows the `set_output_delay` command and options.

Example 6–19. set_output_delay Command

```
set_output_delay
-clock <clock name>
[-clock_fall]
[-rise | -fall]
[-max | -min]
[-add_delay]
[-reference_pin <target>]
<delay value>
<targets>
```

Table 6–15 describes the options for the `set_output_delay` command.

Table 6–15. set_output_delay Command Options (Part 1 of 2)

Option	Description
`-clock` *<clock name>*	Specifies the source clock.
`-clock_fall`	Specifies the required time with respect to the falling edge of the clock.
`-rise | -fall`	Specifies either the rise or fall delay at the port.
`-max | -min`	Specifies the minimum or maximum data arrival time.
`-add_delay`	Adds another delay, but does not replace the existing delays assigned to the port.

Table 6–15. set_output_delay Command Options (Part 2 of 2)

Option	Description
`-reference_pin` <*target*>	Specifies a pin or port in the design from which to determine source and network latencies. Use this option to specify input delays relative to an output port fed by a clock.
`-source_latency_included`	Specifies that the input delay value includes the source latency delay value, and therefore any source clock latency assigned to the clock will subsequently be ignored.
<*delay value*>	Specifies the delay value.
<*targets*>	Specifies the destination ports or pins.



A warning message appears if you specify only a `–max` or `–min` value for the output delay value. The output minimum delay default value is the output maximum delay, and the output maximum delay default value is the output minimum delay, if only one is specified.

The maximum value is used for setup checks, and the minimum value is used for hold checks.

By default, a set of output delays (min/max, rise/fall) is allowed for only one clock, `-clock_fall`, port combination. Specifying an output delay on the same port for a different clock or `-clock_fall` removes any previously set output delays, unless you specify the `-add_delay` option. When you specify the `-add_delay` option, the worst-case values are used.

The `-rise` and `-fall` options are mutually exclusive, as are the `-min` and `-max` options.

Timing Exceptions

Timing exceptions modify the default analysis that is performed by the Quartus II TimeQuest Timing Analyzer. This section describes the following available timing exceptions:

- False path
- Minimum delays
- Maximum delays
- Multicycle path

Precedence

If a conflict of node names occurs between timing exceptions, the following order of precedence applies:

1. False path
2. Minimum delays and maximum delays
3. Multicycle path

The false path timing exception has the highest precedence. Within each category, assignments to individual nodes have precedence over assignments to clocks. Finally, the remaining precedence for additional conflicts is order-dependent, such that the last assignments overwrite (or partially overwrite) earlier assignments.

False Path

False paths are paths that can be ignored during timing analysis.

Use the `set_false_path` command to specify false paths in the design. Example 6–20 shows the `set_false_path` command and options.

Example 6–20. set_false_path Command

```
set_false_path
[-fall_from <clocks> | -rise_from <clocks> | -from <names>]
[-fall_to <clocks> | -rise_to <clocks> | -to <names>]
[-hold]
[-setup]
[-through <names>]
<delay>
```

Table 6–16 describes the options for the `set_false_path` command.

Table 6–16. set_false_path Command Options (Part 1 of 2)

Option	Description
`-fall_from <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies false path begins at the fall from <*clocks*>.
`-fall_to <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies false path ends at the fall to <*clocks*>.
`-from <names>`	The <*names*> is a collection or list of objects in the design. Specifies false path begins at the <*names*>.
`-hold`	Specifies the false path is valid during the hold analysis only.
`-rise_from <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies false path begins at the rise from <*clocks*>.

Table 6–16. set_false_path Command Options (Part 2 of 2)

Option	Description
`-rise_to` <*clocks*>	The <*names*> is a collection or list of objects in the design. Specifies false path ends at the rise to <*clocks*>.
`-setup`	Specifies the false path is valid during the setup analysis only.
`-through` <*names*>	The <*names*> is a collection or list of objects in the design. Specifies false path passes through <*names*>.
`-to` <*names*>	The <*names*> is a collection or list of objects in the design. Specifies false path ends at <*names*>.
<*delay*>	Specifies the delay value.

When the objects are timing nodes, the false path only applies to the path between the two nodes. When an object is a clock, the false path applies to all paths where the source node (`-from`) or destination node (`-to`) is clocked by the clock.

Minimum Delay

Use the `set_min_delay` command to specify an absolute minimum delay for a given path. The following list shows the `set_min_delay` command and options.

Example 6–21. set_min_delay Command

```
set_min_delay
[-fall_from <clocks> | -rise_from <clocks> | -from <names>]
[-fall_to <clocks> | -rise_to <clocks> | -to <names>]
[-through <names>]
<delay>
```

Table 6–17 describes the options for the `set_min_delay` command.

Table 6–17. set_min_delay Command Options

Option	Description
`-fall_from` <*clocks*>	The <*names*> is a collection or list of objects in the design. Specifies the minimum delay begins at the falling edge of <*clocks*>.
`-fall_to` <*clocks*>	The <*names*> is a collection or list of objects in the design. Specifies the minimum delay ends at the falling of <*clocks*>.
`-from` <*names*>	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the start point of the path.
`-rise_from` <*clocks*>	The <*names*> is a collection or list of objects in the design. Specifies the minimum delay at the rising edge of <*clocks*>.
`rise_to` <*clocks*>	The <names> is a collection or list of objects in the design. Specifies the minimum delay at the rising edge of <*clocks*>.
`-through` <*names*>	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the through point of the path.
`-to` <*names*>	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the end point of the path.
<*delay*>	Specifies the delay value.

If the source or destination node is clocked, the clock paths are taken into account, allowing more or less delay on the data path. If the source or destination node has an input or output delay, that delay is also included in the minimum delay check.

When the objects are timing nodes, the minimum delay applies only to the path between the two nodes. When an object is a clock, the minimum delay applies to all paths where the source node (`-from`) or destination node (`-to`) is clocked by the clock.

You can apply the `set_min_delay` command exception to an output port that does not use a `set_output_delay` constraint. In this case, the setup summary and hold summary report the slack for these paths. Because there is no clock associated with the output port, no clock is reported for these paths and the **Clock** column is empty. In this case, you cannot report timing for these paths.



To report timing using clock filters for output paths with the `set_min_delay` command, you must use the `set_output_delay` command for the output port with a value of 0. You can use an existing clock from the design or a virtual clock as the clock reference in the `set_output_delay` command.

Maximum Delay

Use the `set_max_delay` command to specify an absolute maximum delay for a given path. Example 6–22 shows the `set_max_delay` command and options.

Example 6–22. set_max_delay Command

```
set_max_delay
[-fall_from <clocks> | -rise_from <clocks> | -from <names>]
[-fall_to <clocks> | -rise_to <clocks> | -to <names>]
[-through <names>]
<delay>
```

Table 6–18 describes the options for the `set_max_delay` command.

Table 6–18. or set_max_delay Command Options

Option	Description
`-fall_from` <*clocks*>	The <*names*> is a collection or list of objects in the design. Specifies the maximum delay begins at the falling edge of <*clocks*>.
`-fall_to` <*clocks*>	The <*names*> is a collection or list of objects in the design. Specifies the maximum delay ends at the falling of <*clocks*>.
`-from` <*names*>	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the start point of the path.
`-rise_from` <*clocks*>	The <*names*> is a collection or list of objects in the design. Specifies the maximum delay at the rising edge of <*clocks*>.
`rise_to` <*clocks*>	The <names> is a collection or list of objects in the design. Specifies the maximum delay at the rising edge of <*clocks*>.
`-through` <*names*>	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the thru point of the path
`-to` <*names*>	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the end point of the path
<*delay*>	Specifies the delay value.

If the source or destination node is clocked, the clock paths are taken into account, allowing more or less delay on the data path. If the source or destination node has an input or output delay, that delay is also included in the maximum delay check.

When the objects are timing nodes, the maximum delay only applies to the path between the two nodes. When an object is a clock, the maximum delay applies to all paths where the source node (`-from`) or destination node (`-to`) is clocked by the clock.

You can apply the `set_max_delay` command exception to an output port that does not use a `set_output_delay` constraint. In this case, the setup summary and hold summary report the slack for these paths. Because there is no clock associated with the output port, no clock is reported for these paths and the **Clock** column is empty. In this case, you cannot report timing for these paths.



To report timing using clock filters for output paths with the `set_max_delay` command, you must use the `set_output_delay` command for the output port with a value of 0. You can use an existing clock from the design or a virtual clock as the clock reference in the `set_output_delay` command.

Multicycle Path

By default, the Quartus II TimeQuest Timing Analyzer uses a single-cycle analysis. When analyzing a path, the setup launch and latch edge times are determined by finding the closest two active edges in the respective waveforms. For a hold analysis, the timing analyzer analyzes the path against two timing conditions for every possible setup relationship, not just the worst-case setup relationship. Therefore, the hold launch and latch times may be completely unrelated to the setup launch and latch edges.

A multicycle constraint relaxes setup or hold relationships by the specified number of clock cycles based on the source (`-start`) or destination (`-end`) clock. An end multicycle constraint of 2 extends the worst-case setup latch edge by one destination clock period.

Hold multicycle constraints are based on the default hold position (the default value is 0). An end hold multicycle constraint of 1 effectively subtracts one destination clock period from the default hold latch edge.

Use the set_multicycle_path command to specify the multicycle constraints in the design. Example 6–23 shows the set_multicycle_path command and options.

Example 6–23. set_multicycle_path Command

```
set_multicycle_path
[-end]
[-fall_from <clocks> | -rise_from <clocks> | -from <names>]
[-fall_to <clocks> | -rise_to <clocks> | -to <names>]
[-hold]
[-setup]
[-start]
[-through <names>]
<path multiplier>
```

Table 6–19 describes the options for the set_multicycle_path command.

Table 6–19. set_multicycle_path Command Options

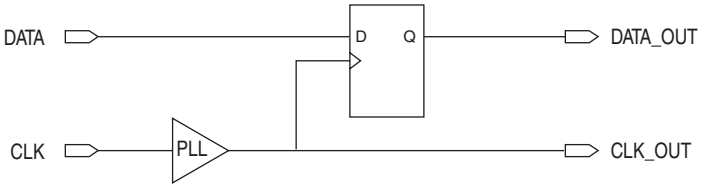
Option	Description
fall_from <*clocks*>	The <*names*> is a collection or list of objects in the design. Specifies the multicycle begins at the falling edge of <*clocks*>.
fall_to <*clocks*>	The <*names*> is a collection or list of objects in the design. Specifies the multicycle ends at the falling of <*clocks*>.
-from <*names*>	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the start point of the path.
-hold|-setup	Specifies the type of multicycle to be applied.
-rise_from <*clocks*>	The <*names*> is a collection or list of objects in the design. Specifies the multicycle at the rising edge of <*clocks*>.
-rise_to <*clocks*>	The <*names*> is a collection or list of objects in the design. Specifies the multicycle ends at the rising edge of <*clocks*>.
-start|-end	Specifies whether the start or end clock acts as the source or destination for the multicycle.
-through <*names*>	The <*names*> is a collection or list of objects in the design. Specifies multicycle passes through <*names*>.
-to <*names*>	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the end point of the path.
<*path multiplier*>	Specifies the multicycle multiplier value.

When the objects are timing nodes, the multicycle constraint only applies to the path between the two nodes. When an object is a clock, the multicycle constraint applies to all paths where the source node (`-from`) or destination node (`-to`) is clocked by the clock.

Clock-as-Data Analysis

The Quartus II TimeQuest Timing Analyzer has the ability to analyze clock paths as data paths. This analysis plays an important part when determining arrival and required times for source synchronous interfaces, or where clock path is used as a data path (for example, a clock captured by a register. Figure 6–28 shows a typical source synchronous interface.

Figure 6–28. Simple Source Synchronous Circuit



To constrain the path from port `clk`, through the PLL, to port `clk_out` you can use a `set_output_delay` to the `clk_out` port, or you can use the `set_max_delay` exception to the `clk_out` port (optionally specifying the PLL clock or PLL output pin as the `-from`). Without the clock as data analysis, this constraint will lose the phase shift associated with the PLL.

With the clock as data analysis the path from port `clk` to port `clk_out` will be analyzed as data path and includes the PLL phase shift. Two paths are reported per analysis: one from the rising edge of the clock source and one from the falling edge of the clock source.

Application Examples

This section describes specific examples for the `set_multicycle_path` command.

Figure 6–29 shows a register-to-register path where the source clock, `src_clk`, has a period of 10 ns and the destination clock, `dst_clk`, has a period of 5 ns.

Figure 6–29. Register-to-Register Path

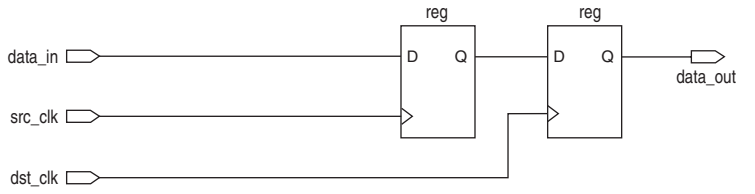
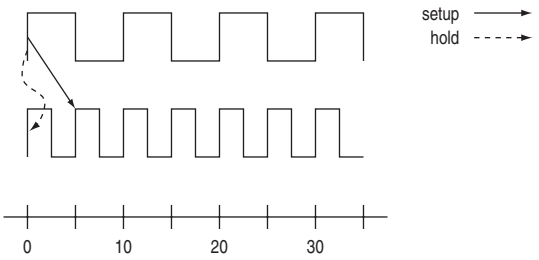


Figure 6–30 shows the respective timing diagrams for the source and destination clocks and the default setup and hold relationships. The default setup relationship is 5 ns and the default hold relationship is 0 ns.

Figure 6–30. Default Setup and Hold Timing Diagram



The default setup and hold relationships can be modified with the `set_multicycle_path` command to accommodate the system requirements.

Table 6–20 shows the commands used to modify either the launch or latch edge times that the TimeQuest Timing Analyzer uses to determine a setup relationship or hold relationship.

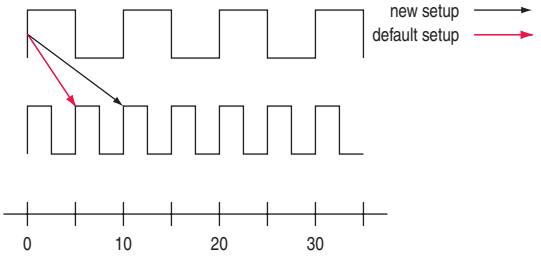
Table 6–20. Commands to Modify Edge Times

Command	Description of Modification
`set_multicycle_path -setup -end`	Latch edge time of the setup relationship
`set_multicycle_path -setup -start`	Launch edge time of the setup relationship
`set_multicycle_path -hold -end`	Latch edge time of the hold relationship
`set_multicycle_path -hold -start`	Latch edge time of the hold relationship

Figure 6–31 shows the command used to modify the setup latch edge and the resulting timing diagram. The command moves the latch edge time to 10 ns from the default 5 ns.

Figure 6–31. Modified Setup Diagram

```
# latch every 2nd edge
set_multicycle_path -from [get_clocks src_clk] -to [get_clocks dst_clk] -setup -end 2
```



Constraint and Exception Removal

When using the Quartus II TimeQuest Timing Analyzer interactively, it is usually necessary to remove a constraint or exception. In cases where constraints and exceptions either become outdated or have been erroneously entered, the Quartus II TimeQuest Timing Analyzer provides a convenient way to remove them.

Table 6–21 lists commands that allow you to remove constraints and exceptions from a design.

Table 6–21. Constraint and Exception Removal

Command	Description
`remove_clock [-all] [<clock list>]`	Removes any clocks specified by <*clock list*> that have been previously created. The -all option removes all declared clocks.
`remove_clock_groups -all`	Removes all clock groups previously created. Specific clock groups cannot be removed.
`remove_clock_latency -source <targets>`	Removes the clock latency constraint from the clock specified by <*targets*>.
`remove_clock_uncertainty -from <from clock> -to <to clock>`	Removes the clock uncertainty constraint from <*from clock*> to <*to clock*>.
`remove_input_delay <targets>`	Removes the input delay constraint from <*targets*>.
`remove_output_delay <targets>`	Removes the output delay constraint from <*targets*>.
`reset_design`	Removes all constraints and exceptions in the design.

Timing Reports

The Quartus II TimeQuest Timing Analyzer provides real-time static timing analysis result reports. Reports are generated only when requested. Each report can be customized to display specific timing information, excluding those fields not required.

This section describes the various report generation commands supported by the Quartus II TimeQuest Timing Analyzer.

report_timing

Use the `report_timing` command to generate a setup, hold, recovery, or removal report. Example 6–24 shows the `report_timing` command and options.

Example 6–24. report_timing Command

```
report_timing
[-append]
[-detail <summary|path_only|path_and_clock|full_path>]
[-from <names>]
[-to <names>]
[-file <name>]
[-fall_from_clock <names> | -rise_from_clock <names> -from_clock <names>]
[-hold]
[-less_than_slack <slack limit>]
[-npaths <number>]
[-nworst <number>]
[-recovery]
[-removal]
[-setup]
[-stdout]
[-through <names>]
[-through <names>]
[-to_clock <names>]
[-panel_name <name>]
[-fall_to_clock <names> | -rise_to_clock <names>]
```

Table 6–22 describes the options for the `report_timing` command.

Table 6–22. report_timing Command Options (Part 1 of 2)

Option	Description
`-append`	Specifies that the current report be appended to the file specified by the `-file` option.
`-file <name>`	Indicates that the current report is written to the file <*name*>.

Table 6–22. report_timing Command Options (Part 2 of 2)

Option	Description
`-detail` <*summary* | *path_only* | *path_and_clock* | *full_path*>	Specifies whether or not the clock path detail is reported. **Path Only**: Clock network delay is lumped together **Summary**: Lists each individual path **Path and Clock**: Clock network delay is shown in detail **Full Path**: More clock network details, in particular for generated clocks
`-fall_from_clock` <*names*>	Specifies the falling edge of the <*names*> from the source register to be analyzed. The options `from_clock`, `fall_from_clock`, and `rise_from_clock` are mutually exclusive.
`-fall_to_clock` <*names*>	Specifies the falling edge of the <*names*> to the destination register to be analyzed. The options `to_clock`, `fall_to_clock`, and `rise_to_clock` are mutually exclusive.
`-file` <*names*>	Indicates that the current report is written to the file <*name*>.
`-hold`	Specifies a clock hold analysis.
`-less_than_slack` <*slack limit*>	Limits the paths to be reported to those the <*slack limit*> value.
`-npaths` <*number*>	Specifies the number of paths to report.
`-nworst` <*number*>	Restricts the number of paths per endpoint.
`-panel_name` <*names*>	Specifies the name of the panel in the **Reports** pane.
`-recovery`	Specifies a recovery analysis.
`-removal`	Specifies a removal analysis.
`-rise_from_clock` <*names*>	Specifies the rising edge of the <*names*> from the source register to be analyzed. The options `from_clock`, `fall_from_clock`, and `rise_from_clock` are mutually exclusive.
`-rise_to_clock` <*names*>	Specifies the rising edge of the <*names*> to the destination register to be analyzed. The options `to_clock`, `fall_to_clock`, and `rise_to_clock` are mutually exclusive.
`-setup`	Specifies a clock setup analysis.
`-stdout`	Indicates the report be sent to stdout.
`-through` <*names*>	Specifies the through node for the analysis.
`-to` <*names*>	Specifies the to node for the analysis.
`-to_clock` <*names*>	Specifies the destination clock for the analysis.
`-panel_name` <*names*>	Sends the results to the panel and specifies the name of the new panel.

Example 6–25 shows a sample report that results from typing the following command:

```
report_timing -from_clock clk_async -to_clock clk_async -setup -npaths 1 ↵
```

Example 6–25. Sample report_timing Report

```
    Info:
============================================================================
    Info: To Node      : dst_reg
    Info: From Node    : src_reg
    Info: Latch Clock  : clk_async
    Info: Launch Clock : clk_async
    Info:
    Info: Data Arrival Path:
    Info:
    Info: Total (ns)  Incr (ns)     Type  Node
    Info: ==========  ========= ==  ====  =========================
    Info:      0.000      0.000           launch edge time
    Info:      2.237      2.237  R        clock network delay
    Info:      2.410      0.173     uTco  src_reg
    Info:      2.410      0.000 RR  CELL  src_reg|regout
    Info:      3.407      0.997 RR    IC  dataout|datain
    Info:      3.561      0.154 RR  CELL  dst_reg
    Info:
    Info: Data Required Path:
    Info:
    Info: Total (ns)  Incr (ns)     Type  Node
    Info: ==========  ========= ==  ====  =========================
    Info:     10.000     10.000           latch edge time
    Info:     11.958      1.958  R        clock network delay
    Info:     11.610     -0.348     uTsu  dst_reg
    Info:
    Info: Data Arrival Time  :     3.561
    Info: Data Required Time :    11.610
    Info: Slack              :     8.049
    Info: ============================================================
```

The `report_timing` command generates a report of the specified analysis type—either setup, hold, recovery, or removal. Each report contains various columns for the data arrival times and data required time, specifically:

- Total
- Incr
- RF
- Type
- Fanout
- Location
- Element

Each of the four column descriptions are described in the Table 6–23.



All columns appear only when a report panel is created. If the `report_timing` output is directed to a file or the console, only the Total, Incr, RF, Type and Node columns appear.

Table 6–23. Timing Report Data

Column Name	Description
Total	Shows the accumulated time delay
Incr	Shows the increment in delay
RF	Shows the input and output transition of the element; this can be one of the following: R, F, RR, RF, FR, FF
Type	Shows the node type; refer to Table 6–24 of a description of the various node types
Fanout	Shows the number of fan-outs of the element
Location	Shows the location of the element in the FPGA
Element	Shows the name element

Table 6–24 provides a description of the possible node Type in the `report_timing` reports.

Table 6–24. Type Description

Type Name	Description
CELL	Indicates the element is either a register or a combinational element in the FPGA; the CELL can be a register in the ALM, memory blocks, or DSP blocks
COMP	Indicates the PLL clock network compensation delay
IC	Indicates the element is an interconnect delay
μt_{CO}	Indicates the element's micro clock-to-out time
μt_{SU}	Indicates the element's micro setup time
μ_{tH}	Indicates the element's micro hold time
i_{EXT}	Indicates the element's external input delay time
o_{EXT}	Indicates the element's external output delay time

report_clock_transfers

Use the `report_clock_transfers` command to generate a report that details all clock-to-clock transfers in the design. A clock-to-clock transfer is reported if a path exists between two registers that are clocked by two different clocks. Information such as the number of destinations and sources is also reported.

Use the `report_clock_transfers` command to generate a setup, hold, recovery, or removal report.

Example 6–26 shows the `report_clock_transfers` command and options.

Example 6–26. report_clock_transfers Command

```
report_clock_transfers
[-append]
[-file <name>]
[-hold]
[-setup]
[-stdout]
[-recovery]
[-removal]
[-panel_name <name>]
```

Table 6–25 describes the options for the `report_clock_transfers` command.

Table 6–25. report_clock_transfers Command Options

Option	Description
`-append`	Specifies that the current report be appended to the file specified by the `-file` option.
`-file <name>`	Indicates that the current report is written to the file <*name*>.
`-hold`	Creates a clock transfer summary for hold analysis.
`-setup`	Creates a clock transfer summary for setup analysis.
`-stdout`	Indicates the report be sent to `stdout`.
`-recovery`	Creates a clock transfer summary for recovery analysis.
`-removal`	Creates a clock transfer summary for removal analysis.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.

report_clocks

Use the report_clocks command to generate a report that details all clocks in the design. The report contains information such as type, period, waveform (rise and fall), and targets for all clocks in the design.

Example 6–27 shows the report_clocks command and options.

Example 6–27. report_clocks Command

```
report_clocks
[-append]
[-desc]
[-file <name>]
[-stdout]
[-panel_name <name>]
```

Table 6–26 describes the options for the report_clocks command.

Table 6–26. report_clocks Command Options

Option	Description
-append	Specifies that the current report be appended to the file specified by the -file option.
-file <*name*>	Indicates that the current report is written to the file <*name*>.
-desc	Specifies the clock names to sort in descending order. The default is ascending order.
-stdout	Indicates the report be sent to stdout.
-panel_name <*name*>	Sends the results to the panel and specifies the name of the new panel.

report_min_pulse_width

A minimum pulse width checks that a clock high or low pulse is sustained enough to recognize an actual change in the clock signal. A failed minimum pulse width check indicates that the register may not recognize the clock transition. Use the report_min_pulse_width command to generate a report that details the minimum pulse width for all clocks in the design. The report contains information for high and low pulses for all clocks in the design.

The report_min_pulse_width command also reports minimum period checks for RAM and DSP, as well as I/O edge rate limits for input and output clock ports. For output ports, the port must either have a clock (or generated clock) assigned to it or used as the -reference_pin for an input/output delays.

The report_min_pulse_width command checks the I/O edge rate limits, but does not always perform the check for output clock ports. For the report_min_pulse_width command to check the I/O edge rate limits for output clock ports the output clock port must:

- Have a clock or generated clock constraint assigned to it

  or

- Be used as a -reference_pin for an Input or Output delay constraint

Each register in the design is reported twice per clock that clocks the register: once for the high pulse and once for the low pulse. Example 6–28 shows the report_min_pulse_width command and options.

Example 6–28. report_min_pulse_width Command

```
report_min_pulse_width
[-append]
[-file <name>]
[-nworst <number>]
[-stdout]
[<targets>]
[-panel_name <name>]
```

Table 6–27 describes the options for the report_min_pulse_width command.

Table 6–27. report_min_pulse_width Command Options

Option	Description
-append	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
-file <*name*>	Sends the results to a file.
-nworst <*number*>	Specifies the number of pulse width checks to report. The default is 1.
-stdout	Redirects the output to stdout via messages; only required if another output format, such as a file, has been selected and is also to receive messages.
-targets	Specifies registers.
-panel_name <*name*>	Sends the results to the panel and specifies the name of the new panel.

report_net_timing

Use the `report_net_timing` command to generate a report that details the delay and fan-out information about a net in the design. A net corresponds to a cell's output pin.

Example 6–29 shows the `report_net_timing` command and options.

Example 6–29. report_net_timing Command

```
report_net_timing
[-append]
[-file <name>]
[-nworst_delay <number>]
[-nworst_fanout <number>]
[-stdout]
[-panel_name <name>]
```

Table 6–28 describes the options for the `report_net_timing` command.

Table 6–28. report_net_timing Command Options

Option	Description
`-append`	Specifies that the current report be appended to the file specified by the `-file` option.
`-file` <*name*>	Indicates that the current report is written to the file <*name*>.
`-nworst_delay` <*number*>	Specifies that <*number*> worst net delays be reported.
`-nworst_fanout` <*number*>	Specifies that <*number*> worst net fan-outs be reported.
`-stdout`	Indicates the report be sent to `stdout`.
`-panel_name` <*name*>	Sends the results to the panel and specifies the name of the new panel.

report_sdc

Use the use the report_sdc command to generate a report of all the Synopsys Design Constraints in the project.

Example 6–30 shows the report_sdc command and options.

Example 6–30. report_sdc Command

```
report_sdc
[-ignored]
[-append]
[-file]
[-stdout]
[-panel_name <name>]
```

Table 6–29 describes the options for the report_sdc command.

Table 6–29. report_sdc Command Options

Option	Description
-ignored	Reports assignments that were ignored.
-append	Specifies that the current report be appended to the file specified by the -file option.
-file	Indicates that the current report is written to the file <*name*>.
-stdout	Indicates that the report is sent to stdout.
-panel_name <*name*>	Sends the results to the panel and specifies the name of the new panel.

report_ucp

Use the report_ucp command to generate a report of all unconstrained paths in the design.

Example 6–31 shows the report_ucp command and options.

Example 6–31. report_ucp Command

```
report_ucp
[-append]
[-file <name>]
[-hold]
[-setup]
[-stdout]
[-summary]
[-panel_name <name>]
```

Table 6–30 describes the options for the report_ucp command.

Table 6–30. Option Descriptions for report_ucp

Option	Description
-append	Specifies that the current report be appended to the file specified by the -file option.
-file <*name*>	Indicates that the current report is written to the file <*name*>.
-hold	Report all unconstrained hold paths.
-setup	Report all unconstrained setup paths.
-stdout	Indicates the report be sent to stdout.
-summary	Generates only the summary panel.
-panel_name <*name*>	Sends the results to the panel and specifies the name of the new panel.

Table 6–31 summarizes all reporting commands available in the Quartus II TimeQuest Timing Analyzer.

Table 6–31. Reports from the Tasks Pane and Tcl Commands (Part 1 of 2)

Task Pane Report	Tcl Command	Description
Report Setup Summary	create_timing_summary -setup	Generates a clock setup summary for all defined clocks.
Report Hold Summary	create_timing_summary -hold	Generates a clock hold summary for all defined clocks.
Report Recovery Summary	create_timing_summary -recovery	Generates a clock recovery summary for all defined clocks.
Report Removal Summary	create_timing_summary -removal	Generates a clock removal summary for all defined clocks.
Report Clocks	report_clocks	Generates a clock summary for all defined clocks.
Report Clock Transfers	report_clock_transfers	Generates a clock transfer summary for all clock-to-clock transfers in the design.
Report SDC	report_sdc	Generates a summary of all SDC file commands read.
Report Unconstrained Paths	report_ucp	Generates a summary of all unconstrained paths in the design.
Report Timing	report_timing	Generates a detailed summary for specific paths in the design.
Report Net Timing	report_net_timing	Generates a detailed summary for specific nets in the design.

Table 6–31. Reports from the Tasks Pane and Tcl Commands (Part 2 of 2)

Task Pane Report	Tcl Command	Description
Report Minimum Pulse Width	`report_min_pulse_width`	Generates a detailed summary for specific registers in the design.
Create Slack Histogram	`create_slack_histogram`	Generates a detailed histogram for a specific clock in the design.

report_path

Use the `report_path` command to report the longest delay paths and the corresponding delay value.

Example 6–32 shows the `report_path` command and options.

Example 6–32. report_path Command

```
report_path
[-append]
[-file <name>]
[-from <names>]
[-npaths <number>]
[-stdout]
[-through]
[-to <names>]
[-panel_name <name>]
```

Table 6–32 describes the options for the `report_path` command.

Table 6–32. report_path Command Options

Option	Description
`-append`	Specifies that the current report be appended to the file specified by the `-file` option.
`-file` *<name>*	Indicates that the current report is written to the file *<name>*.
`-from` *<names>*	Specifies the source node for the analysis.
`-npaths` *<number>*	Specifies the number of paths to report.
`-stdout`	Indicates the report be sent to `stdout`.
`-through` *<name>*	Specifies the through node for the analysis.
`-to` *<names>*	Specifies the destination node for the analysis.
`-panel_name` *<name>*	Sends the results to the panel and specifies the name of the new panel.

report_datasheet

Use the report_datasheet command to generate a datasheet report which summarizes the timing characteristics of the entire design. It reports setup (tsu), hold (th), clock-to-output (tco), minimum clock-to-output (mintco), propagation delay (tpd), and minimum propagation delay (mintpd) times. Example 6–33 shows the report_datasheet command and options.

Example 6–33. report_datasheet Command

```
report_datasheet
[-append]
[-file <name>]
[-stdout]
[panel_name <name>]
```

Table 6–33 describes the options for the report_datasheet command.

Table 6–33. report_datasheet Command Options

Option	Description
-append	Specifies that the current report be appended to the file specified by the -file option.
-file <*name*>	Indicates that the current report is written to the file <*name*>.
-stdout	Indicates the report be sent to stdout.
-panel_name <*name*>	Sends the results to the panel and specifies the name of the new panel.

The delays are reported with respect to a base clock or port for which they are relevant. If there is a case where there are multiple paths for a clock, the maximum delay of the longest path is taken for the tsu, th, tco, and tpd, and the minimum delay of the shortest path is taken for mintco and mintpd.

report_rskm

Use the `report_rskm` command to generate a report that details the receiver skew margin for LVDS receivers.

Example 6–34 shows the `report_rskm` command and options.

Example 6–34. report_rskm Command

```
report_rskm
[-append]
[-file <name>]
[-panel_name <name>]
[-stdout]
```

Table 6–34 describes the options for the `report_rskm` command.

Table 6–34. report_rskm Command Options

Type Name	Description
`-append`	Specifies that the current report be appended to the file specified by the `-file` option.
`-file <name>`	Indicates that the current report is written to the file <*name*>.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.
`-stdout`	Indicates the report be sent to `stdout`.

report_tccs

Use the `report_tccs` command to generate a report that details the channel-to-channel skew margin for LVDS transmitters.

Example 6–35 shows the `report_tccs` command and options.

Example 6–35. report_tccs Command

```
report_tccs
[-append]
[-file <name>]
[-panel_name <name>]
[-quiet]
[-stdout]
```

Table 6–35 describes the options for the report_tccs command.

Table 6–35. report_tccs Command Options

Type Name	Description
-append	Specifies that the current report be appended to the file specified by the -file option.
-file <*name*>	Indicates that the current report is written to the file <*name*>.
-panel_name <*name*>	Sends the results to the panel and specifies the name of the new panel.
-quiet	Specifies that nothing will be printed if there are no LVDS receivers in the design.
-stdout	Indicates the report be sent to stdout.

report_path

Use the report_path command to generate a report that details the longest delay paths between any two arbitrary keeper nodes.

Example 6–36 shows the report_path command and options.

Example 6–36. report_path Command

```
report_path
[-append]
[-file <name>]
[-from <names>]
[-min_path]
[-npaths <number>]
[-nworst <number>]
[-panel_name <name>]
[-stdout]
[-summary]
[-through <names>]
[-to <names>]
```

Table 6–36 describes the options for the report_path command.

Table 6–36. report_path Command Options (Part 1 of 2)

Type Name	Description
-append	Specifies that the current report be appended to the file specified by the -file option.
-file <*name*>	Indicates that the current report is written to the file <*name*>.

Table 6–36. report_path Command Options (Part 2 of 2)

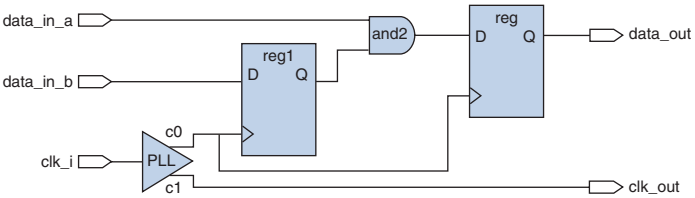
Type Name	Description
`-from <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the start point of the path.
`-min_path`	Displays the minimum delay paths.
`-npaths <number>`	Specifies the number of paths to report.
`-nworst <number>`	Specifies the maximum number of paths to report for each endpoint.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.
`-stdout`	Indicates the report be sent to `stdout`.
`-summary`	Creates a single table with a summary of each path found.
`-through <names>`	The <*names*> is a collection or list of objects in the design. Specifies false path passes through <*names*>.
`-to <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the end point of the path.



The delay path reported cannot pass through a keeper node, for example, a register or port. Instead, the delay path must be from the output pin of a keeper node to the input pin of a keeper node.

Figure 6–32 shows a simple design with a register-to-register path.

Figure 6–32. Simple Register-to-Register Path



Example 6–37 shows the report generated from the following command:

```
report_path -from [get_pins {reg1|regout}] -to [get_pins \
{reg2|datain}] -npaths 1 -panel_name "Report Path" -stdout
```

Example 6–37. report_path from Keeper Output Pin to Keeper Input Pin

```
Info: ===================================================================
Info: From Node : reg1|regout
Info: To Node : reg2|datain
Info:
Info: Path:
Info:
Info: Total (ns) Incr (ns) Type Element
Info: ========== ========= == ==== ===================
Info: 0.000 0.000 reg1|regout
Info: 0.206 0.206 RR IC and2|datae
Info: 0.360 0.154 RR CELL and2|combout
Info: 0.360 0.000 RR IC reg2|datain
Info:
Info: Total Path Delay : 0.360
Info: ===================================================================
```

Example 6–38 shows the report generated from the following command:

```
> report_path -from [get_ports data_in_a] -to [get_pins \
  {reg2|regout}] -npaths 1
```

Example 6–38. report_path from Keeper-to-Keeper Output Pin

```
Info: Report Path: No paths were found
0 0.000
```

No paths were reported in Example 6–38 because the destination passes through an input pin of a keeper node.

check_timing

Use the `check_timing` command to generate a report on any potential problem with the design or applied constraints. Not all `check_timing` results are serious issues, and the results should be examined to see if the results are desired. Example 6–39 shows the `check_timing` command and options.

Example 6–39. check_timing Command

```
check_timing
[-append]
[-file <name>]
[-include <check_list>]
[-stdout]
[-panel_name <name>]
```

Table 6–37 describes the options for the `check_timing` command.

Table 6–37. check_timing Command Options

Option	Description
`-append`	Specifies that the current report be appended to the file specified by the `-file` option.
`-file` <*name*>	Indicates that the current report is written to the file <*name*>.
`-include`	Indicates that a check is to be performed with the <*check_list*>. Refer to Table 6–38 for a list of checks.
`-stdout`	Indicates the report be sent to `stdout`.
`-panel_name` <*name*>	Sends the results to the panel and specifies the name of the new panel.

Table 6–38 describes the possible checks.

Table 6–38. Possible Checks (Part 1 of 2)

Option	Description
`no_clock`	Checks that registers have at least one clock at their clock pin, and that ports determined to be clocks have a clock assigned to them.
`multiple_clock`	Checks that registers have at most one clock at their clock pin. When multiple clocks reach a register's clock pin, both clocks will be used for analysis.
`generated_clock`	Checks that generated clocks are valid. Generated clocks must have a source that is clocked by a valid clock. They must also not depend on each other in a loop (`clk1` cannot have `clk2` as a source if `clk2` already uses `clk1` as a source).
`no_input_delay`	Checks that every input port that is not determined to be a clock has an input delay set on it.
`no_output_delay`	Checks that every output port has an output delay set on it.
`partial_input_delay`	Checks that input delays are complete. Makes sure that input delays have a rise-min, fall-min, rise-max, and fall-max portion set.
`partial_output_delay`	Checks that output delays are complete. Makes sure that output delays have a rise-min, fall-min, rise-max, and fall-max portion set.
`reference_pin`	Checks if reference pins specified in `set_input_delay` and `set_output_delay` using the `reference_pin` option are valid. A `reference_pin` is valid if the clock option specified in the same `set_input_delay`/`set_output_delay` command matches the clock that is in the direct fan-in of the `reference_pin`. Being in the direct fan-in of the `reference_pin` means that there must be no keepers between the clock and the `reference_pin`.

Table 6–38. Possible Checks (Part 2 of 2)

Option	Description
latency_override	Checks if the clock latency constraint that is set on a port or pin overrides the more generic clock latency set on a clock. Clock latency can be set on a clock, where the latency applies to all keepers clocked by the clock, whereas clock latency can also be set on a port or pin, where the latency applies to registers in the fan-out of the port or pin.
loops	Checks that there are no strongly connected components in the design. These loops prevent a design from being properly analyzed. Indicates that loops exist but were marked so that they would not be traversed.
latches	Checks if there are latches in the design. The Quartus II TimeQuest Timing Analyzer warns the user that the latches exist and cannot be properly analyzed.

Example 6–40 shows how the check_timing command can be used.

Example 6–40. The check_timing Command

```
# Constrain design
create_clock -name clk -period 4.000 -waveform { 0.000 2.000 } \
   [get_ports clk]
set_input_delay -clock clk2 1.5 [get_ports in*]
set_output_delay -clock clk 1.6 [get_ports out*]
set_false_path -from [get_keepers in] -through [get_nets r1] -to \
   [get_keepers out]

# Check if there were any problems
check_timing -include {loops latches no_input_delay partial_input_delay}
```

report_clock_fmax_summary

Use the report_clock_fmax_summary to report potential f_{MAX} for every clock in the design, regardless of the user-specified clock periods. f_{MAX} is only computed for paths where the source and destination registers or ports are driven by the same clock. Paths of different clocks, including generated clocks, are ignored. For paths between a clock and its inversion, f_{MAX} is computed as if the rising and falling edges are scaled along with f_{MAX}, such that the duty cycle (in terms of a percentage) is maintained.

Example 6–41 shows the report_clock_fmax_summary command and options.

Example 6–41. report_clock_fmax_summary Command

```
report_clock_fmax_summary
[-append]
[-file <name>]
[-panel_name <name>]
[-stdout]
```

Table 6–39 describes the options for the report_clock_fmax_summary command.

Table 6–39. report_clock_fmax_summary Command Options

Option	Description
-append	Specifies that the current report be appended to the file specified by the -file option.
-file <*name*>	Indicates that the current report is written to the file <*name*>.
-stdout	Indicates the report be sent to stdout.
-panel_name <*name*>	Sends the results to the panel and specifies the name of the new panel.

create_timing_summary

Reports the worst-case Clock Setup and Clock Hold slacks and endpoint TNS (total negative slack) per clock domain. Total negative slack is the sum of all slacks less than zero for each destination register or port in the clock domain.

Example 6–42 shows the create_timing_summary command and options.

Example 6–42. create_timing_summary Command

```
create_timing_summary
[-append]
[-file <name>]
[-hold]
[-panel_name <name>]
[-recovery]
[-removal]
[-setup]
[-stdout]
```

Table 6–40 describes the options for the create_timing_summary command.

Table 6–40. create_timing_summary Command Options

Option	Description
`-append`	Specifies that the current report be appended to the file specified by the `-file` option.
`-file` <*name*>	Indicates that the current report is written to the file <*name*>.
`-hold`	Generates a clock hold check summary report.
`-panel_name` <*name*>	Sends the results to the panel and specifies the name of the new panel.
`-recovery`	Generates a recovery check summary report.
`-removal`	Generates a removal check summary report.
`-setup`	Generates a clock setup check summary report.
`-stdout`	Indicates the report be sent to `stdout`.

Timing Analysis Features

Multi-Corner Analysis

Multi-corner analysis allows a design to be verified under a variety of operating conditions (voltage, process, and temperature) while performing a static timing analysis on the design.

Use the `set_operating_conditions` command to change the operating conditions of the device used for static timing analysis.

Example 6–43 shows the `set_operating_conditions` command and options.

Example 6–43. set_operating_conditions Command

```
set_operating_conditions
[-model <fast|slow>]
[-speed <speed grade>]
[-temperature <value in °C>]
[-voltage <value in mV>]
[<operating condition Tcl object>]
```

Table 6–41 describes the options for the `report_net_timing` command.

Table 6–41. report_net_timing Command Options

Option	Description
`-model` <*fast* | *slow*>	Specifies the timing model.
`-speed` <*speed grade*>	Specifies the device speed grade.
`-temperature` <*value in °C*>	Specifies the operating temperature.
`-voltage` <*value in mV*>	Specifies the operating voltage.
<*operating condition Tcl object*>	Specifies the operating condition Tcl object that specifies the operating conditions.



If an operating condition Tcl object *is* used, the model, speed, temperature, and voltage options are not required. If an operating condition Tcl object is *not* used, the model must be specified, and the -speed, -temperature, and -voltage options are optional, using the appropriate defaults for the device where applicable.

Table 6–42 shows the available operating conditions that can be set for each device family.

Table 6–42. Device Family Operating Conditions

Device Family	Available Conditions			Operating Condition Tcl Objects
	Model	Voltage (mV)	Temp (°C)	

Stratix III	Slow
	Slow
	Fast
		1100
		1100
		1100
			85
			0
			0
				`slow_1100mv_85c`
				`slow_1100mv_0c`
				`fast_1100mv_0c`

Cyclone III	Slow
	Slow
	Fast
		1200
		1200
		1200
			85
			0
			0
				`slow_1200mv_85c`
				`slow_1200mv_0c`
				`fast_1200mv_0c`

Stratix II	Slow Fast	N/A	N/A	`slow` `fast`
Cyclone II	Slow Fast	N/A	N/A	`slow` `fast`



Use the command `get_available_operating_conditions` to obtain a list of available operating conditions for the target device.

Example 6–44 shows how to set the operating conditions for a Stratix III design to the slow model, 1100 mV, and 85°C.

Example 6–44. Setting Operating Conditions with Individual Values

```
set_operating_conditions -model slow -temperature 85 -voltage 1100
```

Alternatively, the operating conditions in Example 6–44 can be set with the Tcl object as shown in Example 6–45.

Example 6–45. Setting Operating Conditions with a Tcl Object

```
set_operating_conditions slow_1100mv_85c
```

Advanced I/O Timing and Board Trace Model Assignments

The Quartus II TimeQuest Timing Analyzer is able to use Advanced I/O Timing and Board Trace Model assignments to model I/O buffer delays in your design. The Advanced I/O Analysis feature can be turned ON or OFF in the **Settings** dialog box under the **TimeQuest Timing Analyzer** option.

If you turn ON or OFF the **Advanced I/O Timing** or change Board Trace Model assignments and do not recompile before you analyze timing, you must use the `-force_dat` option when you create the timing netlist. Type the following command at the Tcl console of the Quartus II TimeQuest Timing Analyzer:

```
create_timing_netlist -force_dat ↵
```

If you turn ON or OFF the Advanced I/O Timing or change Board Trace Model assignments and do recompile before you analyze timing, you do not need to use the `-force_dat` option when you create the timing netlist. You can create the timing netlist with the `create_timing_netlist` command, or with the **Create Timing Netlist** task in the **Tasks** pane.

For more information about the Advanced I/O Timing feature, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Wildcard Assignments and Collections

To simplify the task of applying constraints to many nodes in a design, the Quartus II TimeQuest Timing Analyzer accepts the "*" and "?" wildcard characters. Use these wildcard characters to reduce the number of individual constraints you must specify in your design.

The "*" wildcard character matches any string. For example, given an assignment made to a node specified as reg*, the Quartus II TimeQuest Timing Analyzer searches for and applies the assignment to all design nodes that match the prefix reg with none, one, or several characters following, such as reg1, reg[2], regbank, and reg12bank.

The "?" wildcard character matches any single character. For example, given an assignment made to a node specified as reg?, the Quartus II TimeQuest Timing Analyzer searches and applies the assignment to all design nodes that match the prefix "reg" and any single character following, for example, reg1, rega, and reg4.

Both the collection commands get_cells and get_pins have three options that allow you to refine searches that include the wildcard character. To refine your search results, select the default behavior, the -hierarchical option, or the -compatibility option.



The pipe character is used to separate one hierarchy level from the next in the Quartus II TimeQuest Timing Analyzer. For example, <*absolute full cell name*>|<*pin suffix*> represents a hierarchical pin name with the "|" separating the hierarchy from the pin name.

When you use the collection commands get_cells and get_pins without an option, the default search behavior is performed on a per-hierarchical level of the pin name, that is, the search is performed level by level. A full hierarchical name may contain multiple hierarchical levels where a "|" is used to separate the hierarchical levels, and each wildcard character represents only one hierarchical level. For example, "*" represents the first hierarchical level and "*|*" represents the first and second hierarchical levels.

When you use the collection commands get_cells and get_pins with the -hierarchical option, a recursive match is performed on the relative hierarchical path name of the form <*short cell name*>|<*pin name*>. The search is performed on the node name, for example, the last hierarchy of the name, and not the hierarchy path. Unlike the default behavior, this option does not limit the search to each hierarchy level represented by the pipe character.



The pipe character cannot be used in the search with the get_cells -hierarchical option. The pipe character can be used with the get_pins collection search.

When you use the collection commands `get_cells` and `get_pins` with the `-compatibility` option, the search performed is similar to that of the Quartus II Classic Timing Analyzer. This option searches the entire hierarchical path, and pipe characters are not treated as special characters.

Assuming the following cells exist in a design:

```
foo
foo|bar
```

and the following pin names:

```
foo|dataa
foo|datab
foo|bar|datac
foo|bar|datad
```

Table 6–43 shows the results of using these search strings.

Table 6–43. Sample Search Strings and Search Results

Search String	Search Result
`get_pins *|dataa`	`foo|dataa`
`get_pins *|datac`	*<empty>*
`get_pins *|*|datac`	`foo|bar|datac`
`get_pins foo*|*`	`foo|dataa, foo|datab`
`get_pins -hierarchical *|*|datac`	*<empty>* *(1)*
`get_pins -hierarchical foo|*`	`foo|dataa, foo|datab`
`get_pins -hierarchical *|datac`	`foo|bar|datac`
`get_pins -hierarchical foo|*|datac`	*<empty>* *(1)*
`get_pins -compatibility *|datac`	`foo|bar|datac`
`get_pins -compatibility *|*|datac`	`foo|bar|datac`

Note to Table 6–43:
(1) Due to the additional `*|*|` in the search string, the search result is *<empty>*.

Resetting a Design

Use the `reset_design` command to remove all timing constraints and exceptions from the design under analysis. The command removes all clocks, generated clocks, derived clocks, input delays, output delays, clock latency, clock uncertainty, clock groups, false paths, multicycle paths, min delays, and max delays.

This command provides a convenient way to return to the initial state of analysis without the need to delete and re-create a new timing netlist.

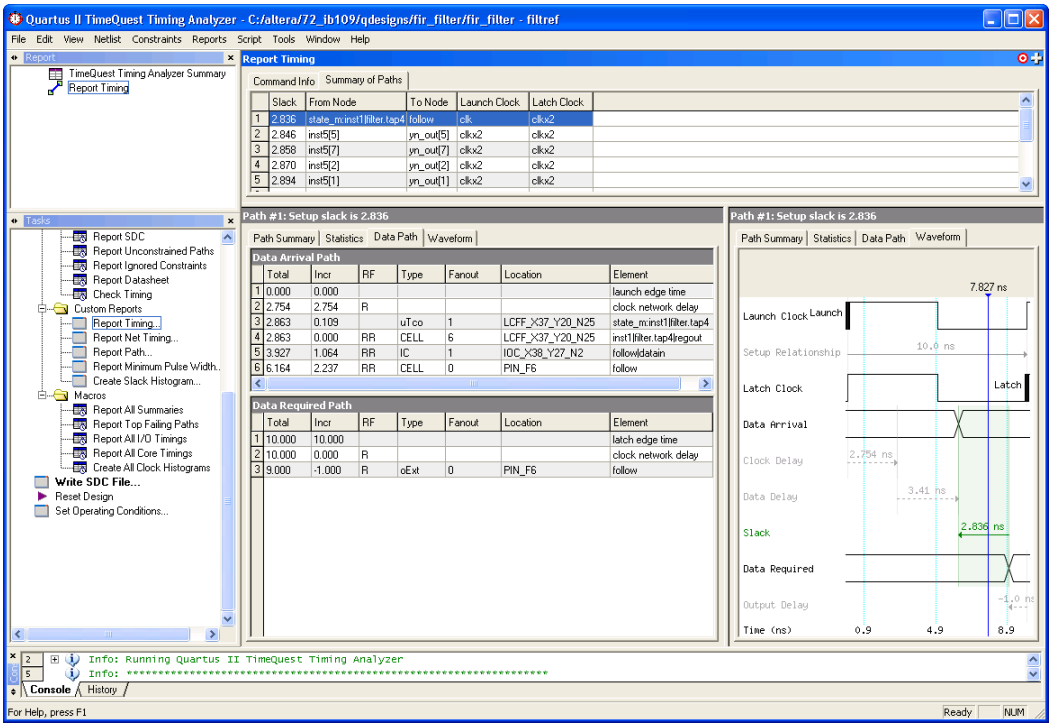
The TimeQuest Timing Analyzer GUI

The Quartus II TimeQuest Timing Analyzer provides an intuitive and easy-to-use GUI that allows you to efficiently constrain and analyze your designs. The GUI consists of the following panes:

- "The Quartus II Software Interface and Options" described on page 6–83
- "View Pane" described on page 6–85
- "Tasks Pane" described on page 6–87
- "Console Pane" described on page 6–90
- "Report Pane" described on page 6–90
- "Constraints" described on page 6–90
- "Name Finder" described on page 6–92
- "Target Pane" described on page 6–94
- "SDC Editor" described on page 6–94

Each pane provides features that enhance productivity (Figure 6–33).

Figure 6–33. The TimeQuest GUI



The Quartus II Software Interface and Options

The Quartus II software allows you to configure various options for the Quartus II TimeQuest Timing Analyzer report generation that are generated in the Compilation Report for the design.

The TimeQuest Timing Analyzer settings, in the **Settings** dialog box, allow you to configure the options shown in Table 6–44.

Table 6–44. The Quartus II TimeQuest Timing Analyzer Settings (Part 1 of 2)

Options	Description
SDC files to include in the project	Adds and removes SDC files associated with the project
Enable Advanced I/O Timing	Generates advanced I/O timing results from board trace models specified for each pin

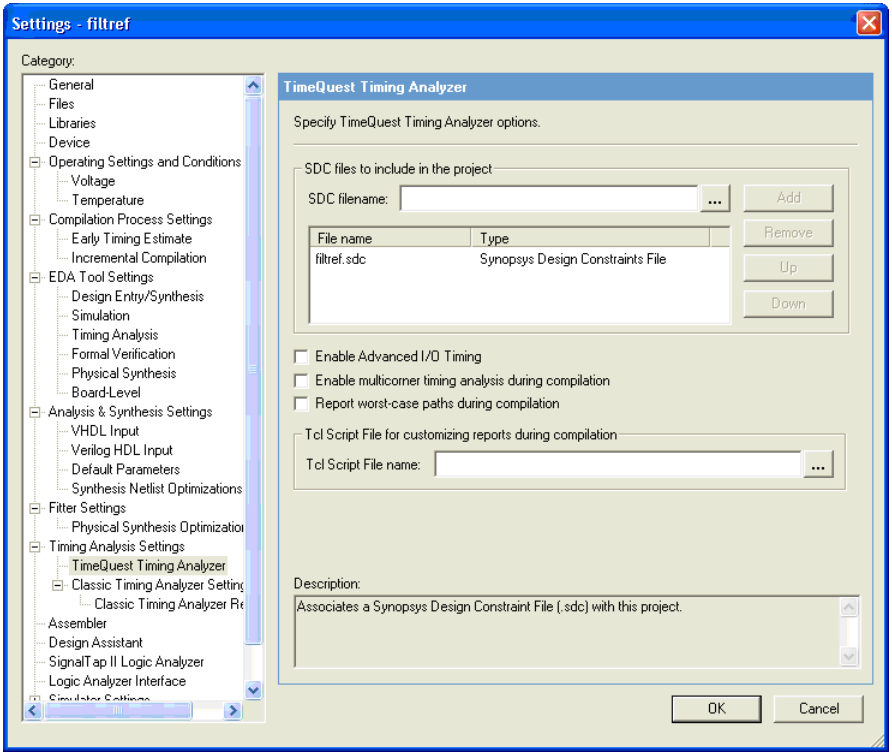
Table 6–44. The Quartus II TimeQuest Timing Analyzer Settings (Part 2 of 2)

Options	Description
Enable multicorner timing analysis during compilation	Generates multiple reports for all available operating conditions of the target device
Report worst-case paths during compilation	Generates worst-case path reports per clock domain
Tcl Script File for customizing report during compilation	Specifies any custom scripts to be sourced for any custom report generation

The options shown in Table 6–44 only control the reports generated in the Compilation Report, and do not control the reports generated in the Quartus II TimeQuest Timing Analyzer.

Figure 6–34 shows the TimeQuest Timing Analyzer setting.

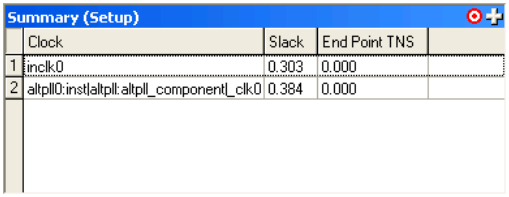
Figure 6–34. TimeQuest Timing Analyzer Settings



View Pane

The **View** pane is the main viewing area for the timing analysis results. Use the **View** pane to view summary reports, custom reports, or histograms. Figure 6–35 shows the **View** pane after you select the Summary (Setup) report from the **Report** pane.

Figure 6–35. Summary (Setup) Report

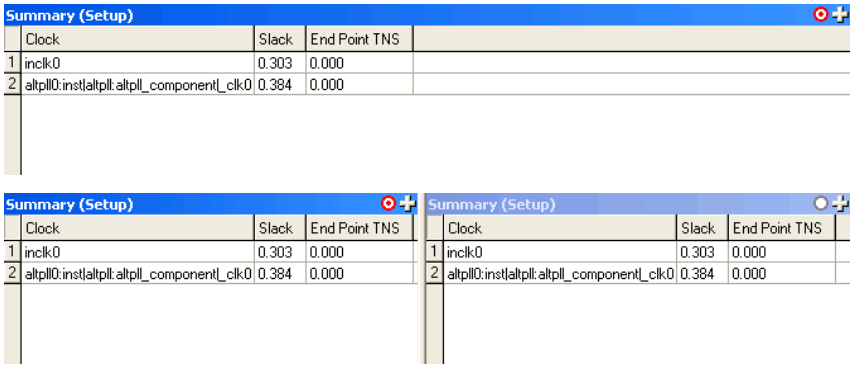


View Pane: Splitting

For the proper analysis of timing results, comparison of multiple reports is extremely important. To facilitate multiple report viewing, the **View** pane supports window splitting. Window splitting divides the **View** pane into multiple windows, allowing you to view different reports side-by-side.

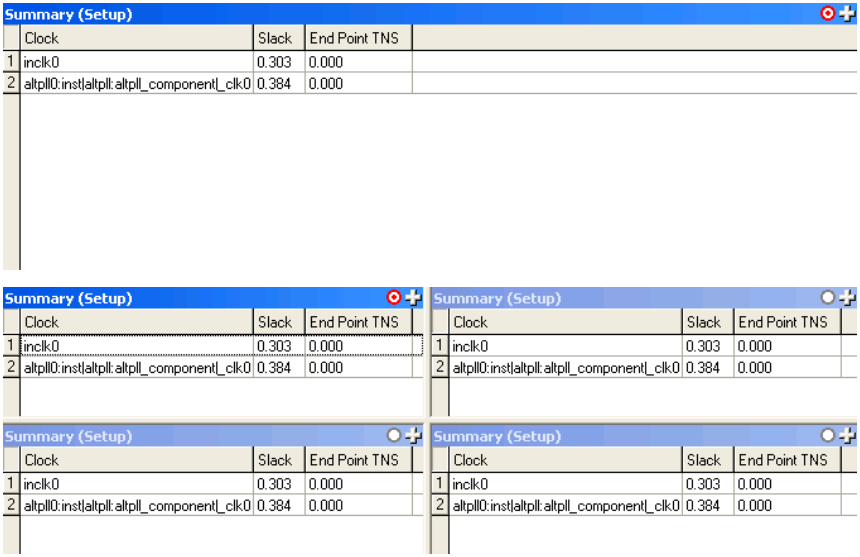
You can split the **View** pane into multiple windows using the split icon located in the upper right hand corner of the **View** pane. Drag the icon in different directions to generate additional window views in the **View** pane. For example, if you drag the split icon to the left, the **View** pane creates a new window to the right of the current window (Figure 6–36).

Figure 6–36. Splitting the View Pane to the Left (Before and After Split Left)



If you drag the split icon diagonally, the **View** pane creates three new windows in the **View** pane (Figure 6–37).

Figure 6–37. Splitting the View Pane Diagonally (Before and After Diagonal Split)

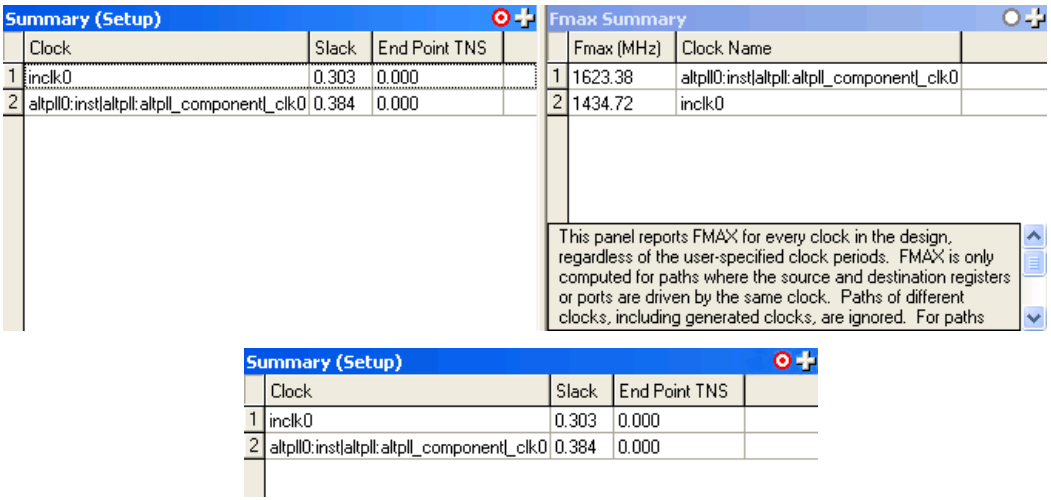


Drag the split icon downward to create a new window directly below the current window.

View Pane: Removing Split Windows

You can remove windows that you create in the **View** pane using the split icon by dragging the border of the window over the window you wish to remove (Figure 6–38).

Figure 6–38. Removing a Split Window (Before and After Split is Removed)



Tasks Pane

Use the **Tasks** pane to access common commands such as netlist setup report generation.

Common commands are located in the **Tasks** pane: **Open Project** and **Write SDC File**, and **Reset Design**. The other commands, including timing netlist setup and the generation of reports, are contained in the following folders:

- **Netlist Setup**
- Reports



Each command in the **Tasks** pane has an equivalent Tcl command that is displayed in the **Console** pane when the command runs.

Opening a Project and Writing a Synopsys Design Constraints File

To open a project in the Quartus II TimeQuest Timing Analyzer, double-click the **Open Project** task. If you launch the Quartus II TimeQuest Timing Analyzer from the Quartus II software GUI, the project opens automatically.

You can add or remove constraints from the timing netlist after the Quartus II TimeQuest Timing Analyzer reads the initial SDC file. After the file is read, the initial SDC file becomes outdated compared to the constraints in the Quartus II TimeQuest Timing Analyzer. Use the **Write SDC File** command to generate an SDC file that is up-to-date and reflects the current state of constraints in the Quartus II TimeQuest Timing Analyzer.

Netlist Setup Folder

The **Netlist Setup** folder contains tasks that are used to set up the timing netlist for timing analysis. The three tasks located in this folder are **Create Timing Netlist**, **Read SDC File**, and **Update Timing Netlist**.

Use the **Create Timing Netlist** task to create a netlist that the Quartus II TimeQuest Timing Analyzer uses to perform static timing analysis. This netlist is used only for timing analysis by the Quartus II TimeQuest Timing Analyzer.



You must always create a timing netlist before you perform static timing analysis with the Quartus II TimeQuest Timing Analyzer.

Use the **Read SDC File** command to apply constraints to the timing netlist. By default, the **Read SDC File** command reads the *<current revision>*.**sdc** file.



Use the `read_sdc` command to read an SDC file that is not associated with the current revision of the design.

Use the **Update Timing Netlist** command to update the timing netlist after you enter constraints. You should use this command if any constraints are added or removed from the design.

Reports Folder

The **Reports** folder contains commands to generate timing summary reports of the static timing analysis results. The nine commands located in this folder are summarized in Table 6–45.

Table 6–45. Reports Folder Commands (Part 1 of 2)

Report Task	Description
Report f_{MAX} Summary	Generates a f_{MAX} summary report for all clocks in the design.
Report Setup Summary	Generates a clock setup summary report for all clocks in the design.

Table 6–45. Reports Folder Commands (Part 2 of 2)

Report Task	Description
Report Hold Summary	Generates a clock hold summary report for all clocks in the design.
Report Recovery Summary	Generates a recovery summary report for all clocks in the design.
Report Removal Summary	Generates a removal summary report for all clocks in the design.
Report Clocks	Generates a summary report of all created clocks in the design.
Report Clock Transfers	Generates a summary report of all clock transfers detected in the design.
Report Minimum Pulse Width	Generates a summary report of all minimum pulse widths in the design.
Report SDC	Generates a summary report of the constraints read from the SDC file.
Report Unconstrained Paths	Generates a summary report of all unconstrained paths in the design.
Report Ignored Constraints	Generates a summary report of all ignored SDC constraints for the design.
Report Datasheet	Generates a datasheet report for the design.

Macros Folder

The **Macros** folder contains commands that perform custom tasks available in the Quartus II TimeQuest Timing Analyzer utility package. These commands are: **Report All Summaries**, **Report Top Failing Paths**, and **Create All Clock Histograms**.

Table 6–46 describes the commands available in the Macros folder.

Table 6–46. Macros Folder Commands

Macro Task	Description
Report All Summaries	This command runs the **Report Setup Summary**, **Report Hold Summary**, **Report Recovery Summary**, **Report Removal Summary**, and **Minimum Pulse Width** commands to generate all summary reports.
Report Top Failing Paths	This command generates a report containing a list of top failing paths.
Create All Clock Histograms	This command runs the **Create Slack Histogram** command to generate a clock histogram for all clocks in the design.
Report All I/O Timings	This command generates a report of all timing paths that start or end at a device port.
Report All Core Timings	This command generates a report of all timing paths that start and end at the device register.

Console Pane

The **Console** pane is both a message center for the Quartus II TimeQuest Timing Analyzer, and an interactive Tcl. The **Console** pane has two tabs: the **Console** tab and the **History** tab. All messages, such as info and warning messages, appear in this pane. Also, the **Console** tab allows you to enter and run Synopsys Design Constraints and Tcl commands. The **Console** tab shows the Tcl equivalent of all commands that you run in the **Tasks** pane. The **History** tab records all the Synopsys Design Constraints and Tcl commands that are run.



To run the commands located in the **History** tab after the timing netlist has been updated, right-click the command, and click **Rerun**.

You can copy Tcl commands from the **Console** and **History** tabs to easily generate Tcl scripts to perform timing analysis.

Report Pane

Use the **Report** pane to access all reports generated from the **Tasks** pane, and by any custom report commands. When you select a report in the **Report** pane, the report is shown in the active window in the **View** pane.



If a report is out-of-date with respect to the current constraints, a "?" icon is shown next to the report.

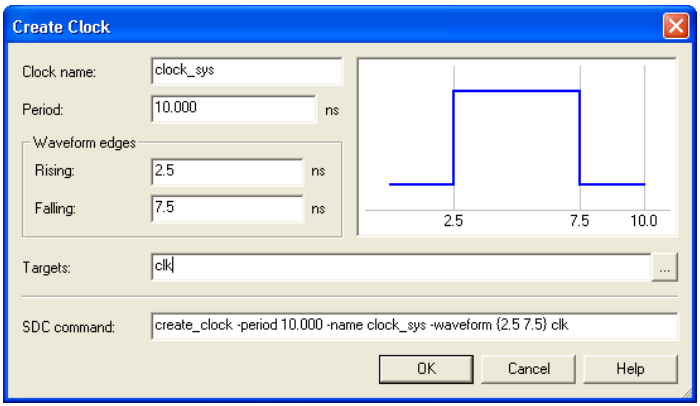
Constraints

Use the Constraints menu to access commonly used constraints, exceptions, and commands. The following commands are available on the Constraints menu:

- **Create Clock**
- **Create Generated Clock**
- **Set Clock Latency**
- **Set Clock Uncertainty**
- **Set Clock Groups**
- **Remove Clock**

For example, use the **Create Clock** dialog box to create clocks in your design. Figure 6–39 shows the **Create Clock** dialog box.

Figure 6–39. Create Clock Dialog Box



The following commands specify timing exceptions, and are available on the Constraints menu:

- **Set False Path**
- **Set Multicycle Path**
- **Set Maximum Delay**
- **Set Minimum Delay**

All the dialog boxes used to specify timing constraints or exceptions from commands have an SDC command field. This field contains the SDC command that is run when you click **OK**.



All commands and constraints created in the Quartus II TimeQuest Timing Analyzer user interface are echoed in the **Console** pane.

The constraints specified with Constraints menu commands are not saved to the current SDC file automatically. You must run the **Write SDC File** command to save your constraints.

The following commands are available on the Constraints menu in the Quartus II TimeQuest Timing Analyzer:

- **Generate SDC File from QSF**
- **Read SDC File**
- **Write SDC File**

The **Generate SDC File from QSF** command runs a Tcl script that converts the Quartus II Classic Timing Analyzer constraints in a QSF file to an SDC file for the Quartus II TimeQuest Timing Analyzer. The file <*current revision*>**.sdc** is created by this command.



For information about the **Generate SDC File from QSF** command, refer to the *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

The **Generate SDC File from QSF** command attempts to convert all timing constraints and exceptions in the QSF file to their equivalent SDC file constraints. However, not all QSF file constraints are convertible to SDC file constraints. Review the SDC file after it is created to ensure that all constraints have been successfully converted.

The **Read SDC File** command reads the <*current revision*>**.sdc** file.

When you select the **Write SDC File** command, an up-to-date SDC file that reflects the current state of constraints in the Quartus II TimeQuest Timing Analyzer is generated.

Name Finder

Use the **Name Finder** dialog box to select the target for any constraints or exceptions in the Quartus II TimeQuest Timing Analyzer GUI. The Name Finder allows you to specify collections, filters, and filter options. The **Collections** field in the **Name Finder** dialog box allows you to specify the type of name to select. To select the type, in the **Collection** list, select the desired collection API including:

- get_cells
- get_clocks
- get_keepers
- get_nets
- get_nodes
- get_pins
- get_ports
- get_registers

For more information about the various collection APIs, refer to "Collections" on page 6–23.

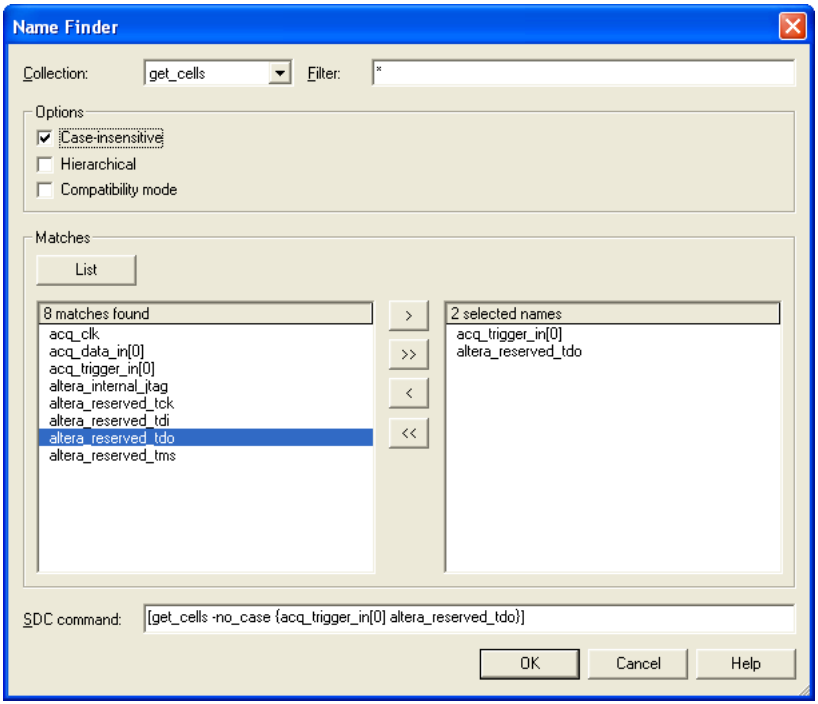
The **Filter** field allows you to filter names based on your own criteria including wildcards characters. You can further refine your results using the following filter options.

- Case-insensitive
- Hierarchical
- Compatibility mode

For more information about the filter options, refer to "Wildcard Assignments and Collections" on page 6–79.

The **Name Finder** dialog box also provides an SDC command field that displays the currently selected name search command. You can copy the value from this field and use it for other constraint target fields. The Name Finder dialog box is shown in Figure 6–40.

Figure 6–40. Name Finder Dialog Box



Target Pane

When using the TimeQuest GUI, you have the ability to split the **View** pane into multiple windows. The splitting feature allows you to display multiple reports in the **View** pane. After splitting the **View** pane, the last active window is updated with any new reports. You can change this behavior by changing the state of each split window. You can do this by clicking on the target circle in the upper right-hand corner (Figure 6–41). Table 6–47 describes the state of each window.

Figure 6–41. Target Pane

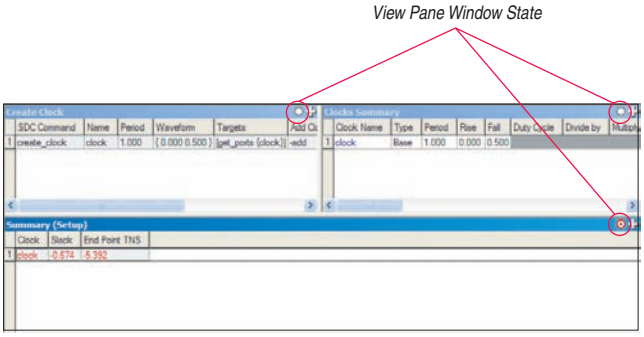


Table 6–47. View Pane Window State

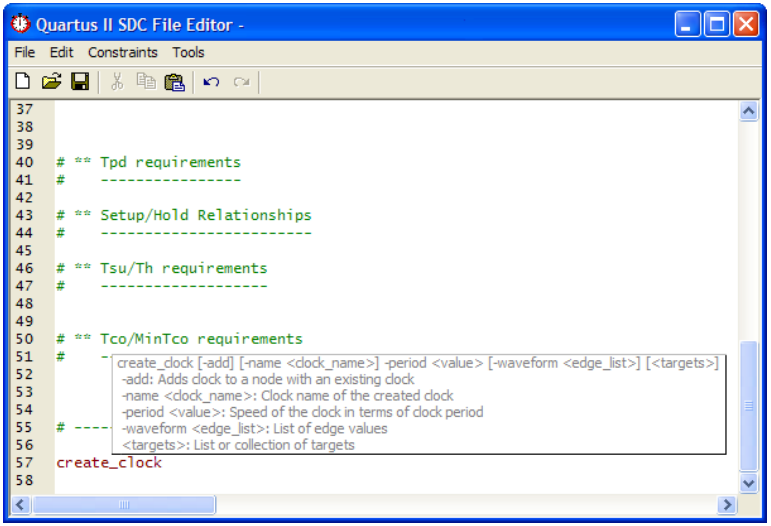
State	Description
Partially Filled Red Circle	Indicates that the active window will display any new reports.
Fully Filled Red Circle	Indicates that the window, independent of it being the active window, will display any new reports.
Empty Circle	Indicates that the window will not display any new reports.

Clicking on the circle in the upper right-hand corner of an active window changes the state of the window.

SDC Editor

The TimeQuest GUI also provides an SDC editor. The SDC editor provides an easy and convenient way to write, edit, and read SDC files directly from the tool. The SDC editor is context sensitive. After an SDC constraint or exception has been entered, a tooltip appears that shows the options and format for the constraint or exception, as shown in Figure 6–42.

Figure 6–42. SDC Editor



The Constraints menu, on the menu bar, allows you to bring up the **Constraints** dialog box. After you have finished entering all required parameters, the SDC is inserted at the current cursor position.

Conclusion

The Quartus II TimeQuest Timing Analyzer caters to the needs of complex designs, resulting in increased productivity and efficiency through its intuitive user interface, support of industry-standard constraints format, and scripting capabilities. The Quartus II TimeQuest Timing Analyzer is a next-generation timing analysis tool that supports the industry-standard SDC format and allows designers to create, manage, and analyze complex timing constraints, and to perform advanced timing verification.

Referenced Documents

This chapter references the following documents:

- *Introduction to Quartus II Manual*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *SDC and TimeQuest API Reference Manual*
- *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*

Document Revision History

Table 6–48 shows the revision history for this chapter.

Table 6–48. Document Revision History

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Updated for the Quartus II software version 7.1, including: • Updated organization flow of the Compilation Flow with TimeQuest Guidelines, Timing Analysis Overview, and Specify Design Timing Requirements sections • Added new information on Clock as Data Analysis	Updated for the Quartus II software version 7.2.

May 2007 v7.1.0	Updated for the Quartus II software version 7.1, including:
	• Added support of `report_path` in "Timing Reports" on page 6–57
	• Added `report_timing` information, especially on page 6-11
	• Added new information under the following headings:
	  ◦ "Derive Clock Uncertainty" on page 6–40
	  ◦ "report_rskm" on page 6–69
	  ◦ "report_tccs" on page 6–69
	  ◦ "report_path" on page 6–70
	• Replaced the "Fast Timing Model Analysis" section with "Multi-Corner Analysis" on page 6–76
	• Performed general 7.1 updates
		Updated for the Quartus II software version 7.1.

March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Updated for the Quartus II software version 6.1, including: • New "Getting Started" section, including descriptions of the Create Clock and Create Generated Clock dialog boxes/commands, sections on Specifying Clock Requirements, Specifying Input and Output Port Requirements, and Reporting • SDC Editor • Usability enhancements to the GUI • Updated SDC support • Numerous changes throughout chapter	Updated for the Quartus II software version 6.1.
July 2006 v6.0.1	Updated for the Quartus II software version 6.0.1: • Fixed typo in report_clock_transfers command on page 6-15.	—
May 2006 v6.0.0	Initial release.	—

7. Switching to the Quartus II TimeQuest Timing Analyzer

QII53019-7.2.0

Introduction

The Quartus II TimeQuest Timing Analyzer provides more powerful timing analysis features than the Quartus II Classic Timing Analyzer. This chapter describes the benefits of switching to the Quartus II TimeQuest Timing Analyzer, the differences between the Quartus II TimeQuest and Quartus II Classic Timing Analyzers, and the process you should follow to switch a design from using the Quartus II Classic Timing Analyzer to the Quartus II TimeQuest Timing Analyzer.

Benefits of Switching to the Quartus II TimeQuest Analyzer

Increasing design complexity requires a timing analysis tool with greater capabilities and flexibility. The Quartus II TimeQuest Timing Analyzer offers the following benefits:

- Industry-standard Synopsys Design Constraint (SDC) support increases productivity.
- Simple, flexible reporting uses industry-standard terminology and makes timing sign-off faster.



For more detailed information about the features and capabilities of the Quartus II TimeQuest Timing Analyzer, refer to the Quartus II TimeQuest Timing Analyzer chapter in volume 3 of the *Quartus II Handbook*.

These features ease constraint and analysis of modern, complex designs. SDC constraints support complex clocking schemes, high-speed interfaces, and more logic. An example includes designs that have multiplexed clocks, regardless of whether they are switched on or off chip. Designs with source-synchronous interfaces, such as DDR memory interfaces, are much simpler to constrain and analyze with the Quartus II TimeQuest Timing Analyzer.

There are three main differences between the Quartus II Classic and Quartus II TimeQuest Timing Analyzers. Unlike the Quartus II Classic Timing Analyzer, the Quartus II TimeQuest Timing Analyzer has the following three benefits:

- All clocks are related by default. (Refer to "Related and Unrelated Clocks" on page 7–13.)
- The default hold multicycle value is zero. (Refer to "Hold Multicycle" on page 7–24.)

- You must constrain all ports and ripple clocks. (Refer to "Automatic Clock Detection" on page 7–19.)

Chapter Contents

"Switching to the Quartus II TimeQuest Analyzer" describes the four-step process you should follow to switch a design to the Quartus II TimeQuest Timing Analyzer.

"Differences Between Quartus II TimeQuest and Quartus II Classic Timing Analyzers" on page 7–5 covers terminology, constraints, clocks, hold multicycle, and other differences.

"Timing Assignment Conversion" on page 7–33 is a comprehensive guide to converting Quartus II Classic QSF timing assignments to Quartus II TimeQuest SDC constraints.

"Conversion Utility" on page 7–55 describes a utility that helps you convert Classic QSF timing assignments toQuartus II TimeQuest SDC constraints.

"Notes" on page 7–68 includes notes about support for specific features in the current version of the Quartus II TimeQuest Timing Analyzer.

Switching to the Quartus II TimeQuest Analyzer

You should use the following process to switch a design from the Quartus II Classic Timing Analyzer to the Quartus II TimeQuest Timing Analyzer. The process is composed of the following steps, which are described in detail in the next sections:

1. Compile your design and perform timing analysis with the Quartus II Classic Timing Analyzer (page 7–2).
2. Create an SDC file that contains timing constraints (page 7–3).
3. Perform timing analysis with the Quartus II TimeQuest Timing Analyzer and examine the reports (page 7–4).
4. Set the default timing analyzer to TimeQuest (page 7–4).

Compile Your Design

To begin, compile your design with the Quartus® II software. You should run the Quartus II Classic Timing Analyzer during compilation because it is easier to convert your assignments to SDC constraints when you create an SDC file. To run the Quartus II Classic Timing Analyzer in the Quartus II GUI, on the Processing menu, click **Start**, then click **Start**

Timing Analyzer. To run the Quartus II Classic Timing Analyzer if you are a command-line user, type `quartus_tan` <*project*> ↵ at a system command prompt.

Create an SDC File

The Quartus II TimeQuest Timing Analyzer supports SDC format constraints. If you are familiar with SDC terminology, you can create an SDC file with any text editor and skip to "Perform Timing Analysis with the Quartus II TimeQuest Timing Analyzer" on page 7–4. Name the SDC file <*revision*>**.sdc** (<*revision*> is the current revision of your project) and save it in your project directory.



Refer to the *SDC and TimeQuest Tcl API Reference Manual* for a TimeQuest SDC command reference.

Alternately, you can use a Quartus II TimeQuest conversion utility to help you convert the timing assignments in an existing QSF file to corresponding SDC constraints.

Conversion Utility

To run the Quartus II TimeQuest conversion utility, click **Generate SDC file from QSF** on the Constraints menu. You can also run the conversion utility by typing either of the following commands at a system command prompt:

✓ `quartus_tan --qsf2sdc` <*project name*> ↵

or

✓ `quartus_sta --qsf2sdc` <*project name*> ↵

The SDC file created by the conversion utility is named <*revision*>**.sdc**.

For information about how to run the Quartus II TimeQuest Timing Analyzer, refer to "Run the Quartus II TimeQuest Analyzer" on page 7–4.



If you use the conversion utility, you must review the SDC file to ensure it is correct and complete, and make changes if necessary. Refer to "Constraint File Priority" on page 7–10 for the recommended way to make changes.

The conversion utility cannot convert some types of Quartus II Classic assignments for the following reasons:

- No corresponding SDC constraint exists

- Multiple SDC constraints are valid, so correct conversion requires knowledge of the intended function of your design

You must manually convert any such assignments based on the guidelines in "Timing Assignment Conversion" on page 7–33.

Perform Timing Analysis with the Quartus II TimeQuest Timing Analyzer

When your SDC file is complete, use the reporting capabilities in the Quartus II TimeQuest Timing Analyzer. If you use the Quartus II TimeQuest GUI, double-click any of the reports listed in the **Task** pane. You can also type commands in the Quartus II TimeQuest Tcl shell to generate reports.

You should also review "Notes" on page 7–68 to ensure the Quartus II TimeQuest Timing Analyzer supports all stages of your design flow.



For complete information about how to use the Quartus II TimeQuest Timing Analyzer, and descriptions of commands and reports, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*, and the *SDC and TimeQuest Tcl API Reference Manual*.

Run the Quartus II TimeQuest Analyzer

If you are using the Quartus II software, to open the Quartus II TimeQuest GUI, on the Tools menu, click **TimeQuest Timing Analyzer**. The Quartus II TimeQuest GUI automatically opens the project you have open in the Quartus II GUI.

If you use the system command prompt to open the Quartus II TimeQuest Timing Analyzer, type `quartus_staw` ↵ to open the Quartus II TimeQuest GUI, or type `quartus_sta -s` ↵ to start the Quartus II TimeQuest Timing Analyzer in Tcl shell mode. Use the **project_open** command to open your project, or, on the File menu, click **Open Project**.

Set the Default Timing Analyzer

To use the Quartus II TimeQuest Timing Analyzer as the default timing analyzer for your project, turn on **Use TimeQuest Timing Analyzer during compilation**. In the Quartus II GUI, on the Assignments menu, click **Settings**, then click the **Timing Analysis Settings** category, and select **Use TimeQuest Timing Analyzer during compilation**. You can make the same setting in your project's QSF file with the following Tcl command:

```
set_global_assignment -name \
USE_TIMEQUEST_TIMING_ANALYZER ON
```

This setting directs the Quartus II software to use the Quartus II TimeQuest Timing Analyzer instead of the Quartus II Classic Timing Analyzer.

The setting to make the Quartus II TimeQuest Timing Analyzer the default Timing Analyzer is specific to each project, so you can decide on a per-project basis whether to use the Quartus II TimeQuest Timing Analyzer or the Quartus II Classic Timing Analyzer.

If you want to use the Quartus II Classic Timing Analyzer instead of the Quartus II TimeQuest timing analyzer, ensure **Use Classic Timing Analyzer during compilation** is selected. You can delete the <*revision*>**.sdc** file, because the Quartus II Classic Timing Analyzer does not use it.

In the Quartus II software, a timing analyzer performs two functions:

- Processing timing constraints and exceptions that affect how your design is placed and routed
- Reporting after place and route is complete so you know whether the design meets timing requirements

Although you can use one timing analyzer to process timing constraints during place and route and the other for reporting, you should use the same timing analyzer for both. The Quartus II Classic Timing Analyzer uses assignments in the QSF file, and theQuartus II TimeQuest Timing Analyzer uses constraints in the SDC file. Any differences between the timing assignments in the two files may cause inconsistent results.

Differences Between Quartus II TimeQuest and Quartus II Classic Timing Analyzers

TheQuartus II TimeQuest Timing Analyzer is different from the Quartus II Classic Timing Analyzer in the following ways:

- Terminology (page 7–5)
- Constraints (page 7–7)
- Clocks (page 7–13)
- Hold Multicycle (page 7–24)
- Fitter Behavior (page 7–27)
- Reporting (page 7–27)
- Scripting API (page 7–32)

Terminology

This section introduces the industry-standard SDC terminology that the Quartus II TimeQuest Timing Analyzer uses.

For more detailed information about this terminology, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Netlist

The Quartus II TimeQuest Timing Analyzer uses SDC naming conventions for netlists. Netlists consist of cells, pins, nets, ports, and clocks.

- Cells are instances of fundamental hardware elements in Altera® FPGAs (such as logic elements, look-up tables, and registers).
- Pins are inputs and outputs of cells.
- Nets are connections between output pins and input pins.
- Ports are top-level module inputs and outputs (device inputs and outputs).
- Clocks are abstract objects outside the netlist.

The terminology of pins and ports is opposite to that of the Quartus II Classic Timing Analyzer. In the Quartus II Classic Timing Analyzer, ports are inputs and outputs of cells, and pins are top-level module inputs and outputs (device inputs and outputs).

Figure 7–1 shows a simple design, and Figure 7–2 shows the Quartus II TimeQuest netlist representation of the design. Netlist elements in Figure 7–2 are labeled to illustrate the SDC terminology.

Figure 7–1. Sample Design

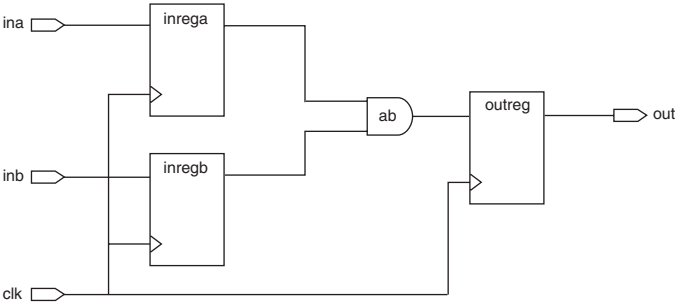
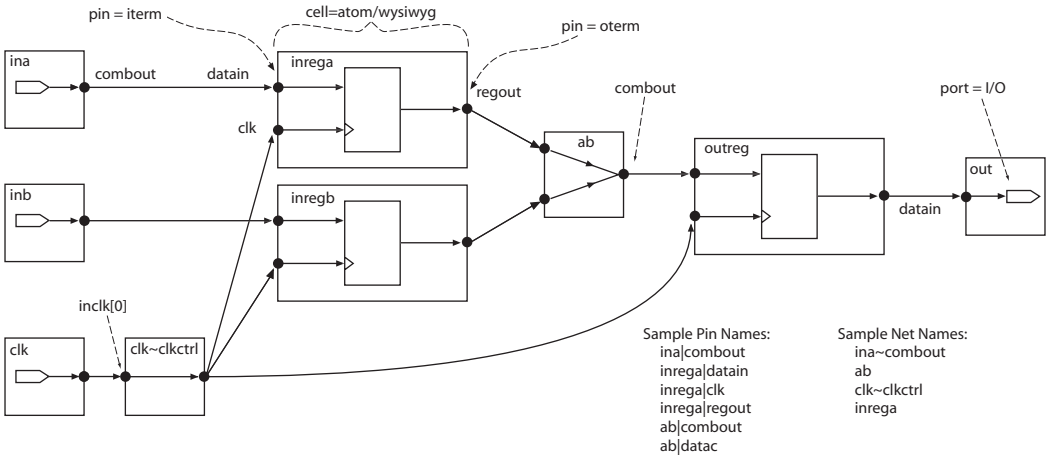


Figure 7–2. Quartus II TimeQuest Timing Analyzer Netlist



Collections

In addition to standard SDC collections, the Quartus II TimeQuest Timing Analyzer supports the following Altera-specific collection types:

- **Keepers**—Non-combinational nodes in a netlist
- **Nodes**—Nodes can be combinational, registers, latches, or ports (device inputs and outputs)
- **Registers**—Registers or latches in the netlist

You can use the **get_keepers**, **get_nodes**, or **get_registers** commands to access these collections.

Constraints

The Quartus II Classic and Quartus II TimeQuest Timing Analyzers store constraints in different files, support different methods for constraint entry, and prioritize constraints differently. The following sections detail these differences.

Constraint Files

The Quartus II TimeQuest Timing Analyzer stores all SDC timing constraints in SDC files. The Quartus II Classic Timing Analyzer stores all timing assignments in your project's Quartus II Settings File (QSF) file. The QSF file contains all your project's assignments and settings except for the Quartus II TimeQuest Timing Analyzer constraints. The

Quartus II TimeQuest Timing Analyzer ignores the timing assignments in your QSF file except when the conversion utility converts Quartus II Quartus II Classic QSF timing assignments to Quartus II TimeQuest SDC constraints. There is no automatic process that keeps timing constraints synchronized between your QSF and SDC files. If you want to keep the constraints synchronized, you must convert them manually.

Constraint Entry

In the Quartus II Classic Timing Analyzer, you enter timing assignments with the Settings dialog box, the Assignment Editor, or with commands in Tcl scripts. The Quartus II TimeQuest Timing Analyzer does not use the Assignment Editor for its constraints, and you cannot use the Assignment Editor to enter SDC constraints. You must use one of the following methods to enter Quartus II TimeQuest constraints:

- Enter constraints at the Tcl prompt in the Quartus II TimeQuest Timing Analyzer
- Enter constraints in an SDC file with a text editor or SDC editor
- Use the constraint entry commands on the Constraints menu in the Quartus II TimeQuest GUI

You can enter timing assignments for the Quartus II Classic Timing Analyzer even if no timing netlist exists for your design. The Quartus II TimeQuest Timing Analyzer requires that a netlist exist for interactive constraint entry. Each Quartus II TimeQuest Timing Analyzer constraint is a Tcl command evaluated in real-time, if entered directly in the Tcl console. As part of this evaluation, the Quartus II TimeQuest Timing Analyzer validates all names. To do this, SDC commands can only be evaluated after a netlist is created. An SDC file can be created at any time using the Quartus II TimeQuest Timing Analyzer or any other text editor, but a netlist is required before an SDC file can be sourced. You must create a timing netlist in the Quartus II TimeQuest Timing Analyzer before you can enter constraints with either of the following interactive methods:

- At the Tcl console of the Quartus II TimeQuest Timing Analyzer
- With commands on the Constraints menu in the Quartus II TimeQuest GUI

If you enter constraints with a text editor separate from the Quartus II TimeQuest Timing Analyzer, no timing netlist is required.

To create a timing netlist in the Quartus II TimeQuest Timing Analyzer, use the **create_timing_netlist** command, or double-click **Create Timing Netlist** in the Task pane of the Quartus II TimeQuest GUI.

If you have never compiled your design, and you want to use the Quartus II TimeQuest Timing Analyzer to enter constraints interactively, you must synthesize your design before you create a timing netlist. To synthesize your design, type `quartus_map` <*project name*> ↵ at a system command prompt, or, if you use the Quartus II GUI, ensure that your project is open, then click **Start** on the Processing menu, and click **Start Analysis and Synthesis**.

To create the netlist, open the Quartus II TimeQuest Timing Analyzer. Then, on the Netlist menu, click **Create Timing Netlist...**, select **Post-map**, and click **OK**. Alternately, type `create_timing_netlist -post_map` ↵ at the Tcl Console.

Time Units

Enter time values are in default time units of nanoseconds (ns) with up to three decimal places. Note that the Quartus II TimeQuest Timing Analyzer does not display the default time unit when it displays time values.

You can specify a different default time unit with the **set_time_format -unit** <*default time unit*> command, or specify another unit when you enter a time value, for example, `300ps`.



Specifying time units with the value is not part of the standard SDC format. This is a Quartus II TimeQuest extension.

You can specify clock constraints with period or frequency in the Quartus II TimeQuest Timing Analyzer. For example, you can use either of the following constraints:

- `create_clock -period 10.000` (assuming default units and decimal places)
- `create_clock -period "100 MHz"`
- `create_clock -period "10 ns"`

MegaCore Functions

If you change any MegaCore function settings and regenerate the core after you convert your timing assignments to SDC constraints, you must manually update the SDC constraints or reconvert your assignments. You must update or reconvert, because changes to MegaCore function settings can affect timing assignments embedded in the hardware description language files of the core. The timing assignments are not converted automatically when the core settings change.



You should make a backup copy of your SDC file before reconverting assignments. If you made changes to the SDC file, you can manually copy the updated MegaCore timing constraints to your SDC file.

Bus Name Format

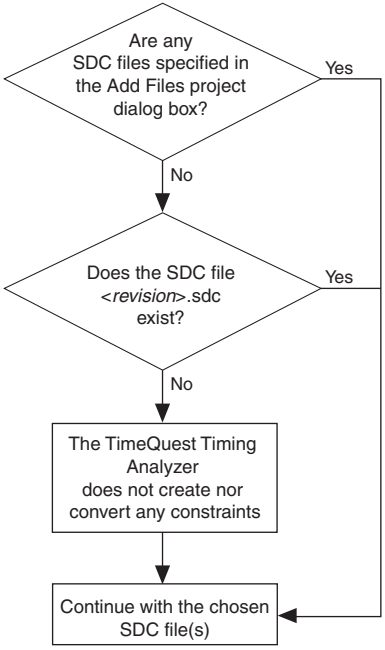
In the Quartus II Classic Timing Analyzer, you can make a timing assignment to all bits in a bus with the bus name (or the bus name followed by an asterisk enclosed in square brackets) as the target. For example, to make an assignment to all bits of a bus called address, use `address` or `address[*]` as the target of the assignment.

In the Quartus II TimeQuest Timing Analyzer, you must use the bus name followed by square brackets enclosing an asterisk, like this: `address[*]`.

Constraint File Priority

The Quartus II TimeQuest Timing Analyzer searches for SDC files with a specific priority, as shown in Figure 7–3.

Figure 7–3. SDC File Search Order



If you specify constraints in multiple SDC files, or if you use a single SDC file with a name other than <*revision*>**.sdc**, you must add the files to your project so the Quartus II TimeQuest Timing Analyzer can find them. If you use the Quartus II software, click **Add/Remove Files in Project** on the Project menu, and add the appropriate SDC files. You can also add SDC files to your project with the following Tcl command in your QSF file, repeated once for each SDC file:

```
set_global_assignment -name SDC_FILE <SDC file name>
```

The Quartus II TimeQuest Timing Analyzer reads constraint files from the files list in the order they are listed, first to last.



If you use an SDC file created by the conversion utility, you should place it before all other SDC files in the list of files. When conflicting constraints apply to the same node, the last constraint has the highest priority. Therefore, SDC files with your additions or changes should be listed after the SDC file created by the conversion utility, so your constraints have higher priority.

Beginning with version 6.1, the Quartus II TimeQuest Timing Analyzer does not run the conversion utility automatically when it cannot find an SDC file according to the priority shown in Figure 7–3. It may prompt you to run the conversion utility from the Constraints menu in the Quartus II TimeQuest GUI.



You must review the SDC file as you would when manually running the conversion utility. Refer to "Reviewing Conversion Results" on page 7–64 for information about how to review the converted constraints.

If no SDC file exists when you run the Quartus II Fitter, and you have turned on **Use TimeQuest Timing Analyzer during compilation**, the Fitter does not create an SDC file automatically, but it attempts to meet a default 1 GHz constraint on all clocks in your design.

Constraint Priority

The Quartus II Classic Timing Analyzer prioritizes assignments based on the specificity of the nodes to which they are assigned. The more specific an assignment is, the higher its priority. The Quartus II TimeQuest Timing Analyzer simplifies these precedence rules. When overlaps occur in the nodes to which the constraints apply, constraints at the bottom of the file take priority over constraints at the top of the file.

As an example, in the Quartus II Classic Timing Analyzer, point-to-point multicycle assignments have higher priority than single point multicycle assignments. The two assignments in Example 7–1 result in a multicycle assignment of 2 between A_reg and all nodes beginning with B, including B_reg. The single point assignment does not apply to paths from A_reg to B_reg, because the specific point-to-point assignment takes priority over the general single point assignment.

Example 7–1. Quartus II Classic Timing Analyzer Multicycle Assignments

```
set_instance_assignment -name MULTICYCLE -from A_reg -to B* 2
set_instance_assignment -name MULTICYCLE -to B_reg 3
```

Example 7–2 shows SDC versions of the Quartus II Classic Timing Analyzer timing assignments above. However, the Quartus II TimeQuest Timing Analyzer evaluates the constraints top to bottom (regardless of point-to-point or single point), so the path from A_reg to B_reg receives a multicycle exception of 3 because it is second in order.

Example 7–2. Quartus II TimeQuest Timing Analyzer Multicycle Exceptions

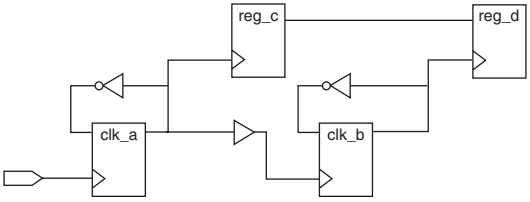
```
set_multicycle_path -from [get_keepers A_reg] -to [get_keepers B*] 2
set_multicycle_path -to [get_keepers B_reg] 3
```

Ambiguous Constraints

Because of new capabilities in the Quartus II TimeQuest Timing Analyzer, some Quartus II Classic assignments can be ambiguous after conversion by the conversion utility. These assignments require manual updating based on your knowledge of your design.

Figure 7–4 shows a ripple clock circuit. The explanation that follows shows an ambiguous constraint for that circuit, and how to edit the constraint to remove the ambiguity in the Quartus II TimeQuest Timing Analyzer.

Figure 7–4. Ripple Clock Circuit



In the Quartus II Classic Timing Analyzer, the following QSF multicycle assignment from clk_a to clk_b with a value of 2 applies to paths transferring data from the clk_a domain to the clk_b domain.

```
set_instance_assignment -name MULTICYCLE -from clk_a -to clk_b 2
```

In Figure 7–4, this assignment applies to the path from reg_c to reg_d. In the Quartus II TimeQuest Timing Analyzer, the use of the clock node names in the following equivalent multicycle exception is ambiguous.

```
set_multicycle_path -setup -from clk_a -to clk_b 2
```

The exception could apply to the path between clk_a and clk_b, or it could apply to paths from one ripple clock domain to the other ripple clock domain (reg_c to reg_d).

The Quartus II TimeQuest exceptions shown in Example 7–3 are not ambiguous because they use collections to explicitly specify the targets of the exception.

Example 7–3. Unambiguous Quartus II TimeQuest Timing Analyzer Exceptions

```
# Applies to path from reg_c to reg_d
set_multicycle_path -setup -from [get_clocks clk_a] \
    -to [get_clocks clk_b] 2
# Applies to path from clk_a to clk_b
set_multicycle_path -setup -from [get_registers clk_a] \
       -to [get_registers clk_b] 2
```

Clocks

The Quartus II Classic and Quartus II TimeQuest Timing Analyzers detect, analyze, and report clocks differently. The following sections describe these differences.

Related and Unrelated Clocks

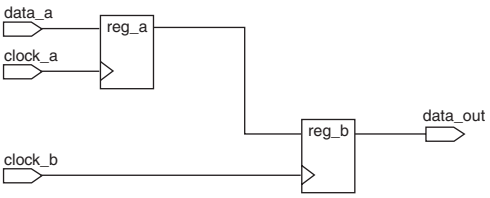
In the Quartus II TimeQuest Timing Analyzer, all clocks are related by default, and you must add assignments to indicate unrelated clocks. However, in the Quartus II Classic Timing Analyzer, all base clocks are unrelated by default. All derived clocks of a base clock are related to each other, but are unrelated to other base clocks and their derived clocks.



You can change the default behavior of the Quartus II Classic Timing Analyzer to treat all clocks as related clocks. On the Assignments menu, click **Timing Analysis Settings**. Click **More Settings** and then select **Cut paths between unrelated clock domains**. Ensure that the setting is off.

Figure 7–5 on page 7–14 shows a simple circuit with a path between two clock domains. The Quartus II TimeQuest Timing Analyzer analyzes the path from `reg_a` to `reg_b` because all clocks are related by default. The Quartus II Classic Timing Analyzer does not analyze the path from `reg_a` to `reg_b` by default.

Figure 7–5. Cross Clock Domain Path



To make clocks unrelated in the Quartus II TimeQuest Timing Analyzer, use the **set_clock_groups** command with the `-exclusive` option. For example, the following command makes `clock_a` and `clock_b` unrelated, so the Quartus II TimeQuest Timing Analyzer does not analyze paths between the two clock domains.

```
set_clock_groups -exclusive -group {clock_a} -group {clock_b}
```

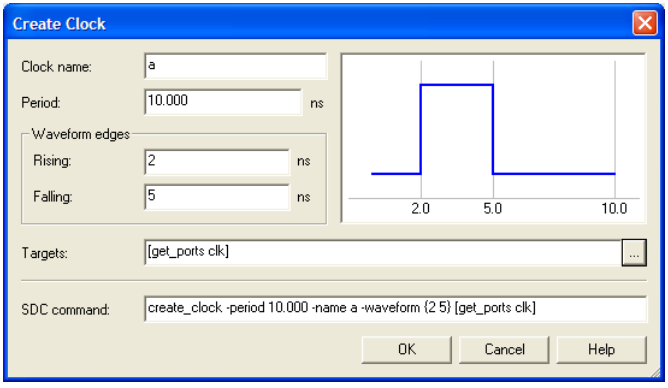
Clock Offset

In the Quartus II TimeQuest Timing Analyzer, clocks can have non-zero values for the rising edge of the waveform, a feature that the Quartus II Classic Timing Analyzer does not support. To specify an offset for your clock, use the waveform option for the **create_clock** command to specify the rising and falling edge times, as shown in this example:

`-waveform {`*<rising edge time>* *<falling edge time>*`}`

Figure 7–6 shows a clock constraint with an offset in the Quartus II TimeQuest Timing Analyzer GUI.

Figure 7–6. Create Clock Screen



Clock offset affects setup and hold relationships. Launch and latch edges are evaluated after offsets are applied. Depending on the offset, the setup relationship can be the offset value, or the difference between the period and offset. You should not use clock offset to emulate latency. You should use the clock latency constraint instead. Refer to "Offset and Latency Example" on page 7–15 for an example that illustrates the different effects of offset and latency.

Clock Latency

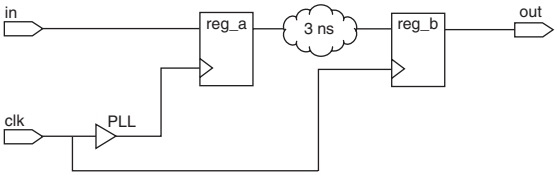
The Quartus II TimeQuest Timing Analyzer does not ignore early clock latency and late clock latency differences when the clock source is the same, as the Quartus II Classic Timing Analyzer does. When you specify latencies, you should take common clock path pessimism into account and use uncertainty to control pessimism differences for clock-to-clock data transfers. Unlike clock offset, clock latency affects skew, and launch and latch edges are evaluated before latencies are applied, so the setup relationship is always equal to the period.

Offset and Latency Example

Figure 7–7 shows a simple register-to-register circuit used to illustrate the different effects of offset and latency. The examples show why you should not use clock offset to emulate clock latency. You should always turn on

the **Enable Clock Latency** option in the Quartus II Classic Timing Analyzer. This option is in the **More Settings** box of the **Timing Settings** dialog box.

Figure 7–7. Simple Circuit for Offset and Latency Examples

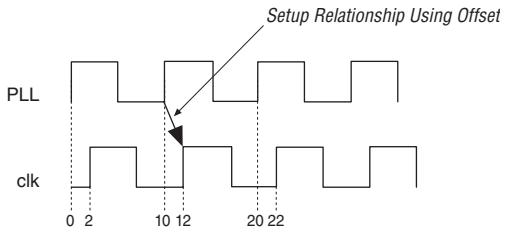


The period for `clk` is 10 ns, and the period for the PLL output is 10 ns. The PLL compensation value is –2 ns. The network delay from the PLL to `reg_a` equals the network delay from `clk` to `reg_b`. Finally, the delay from `reg_a` to `reg_b` is 3 ns.

Clock Offset Scenario

Treat the PLL compensation value of –2 ns as a clock offset of –2 ns with a clock skew of 0 ns. Launch and latch edges are evaluated after offsets are applied, so the setup relationship is 2 ns (Figure 7–8).

Figure 7–8. Setup Relationship Using Offset



Equation 1 shows how to calculate the slack value for the path from reg_a to reg_b.

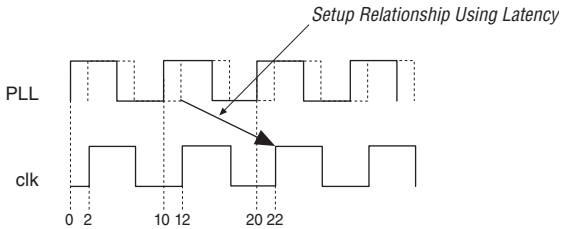
(1)
$$\text{slack} = \text{clock arrival} - \text{data arrival}$$
$$\text{slack} = \text{setup relationship} + \text{clock skew} - \text{reg_to_reg delay}$$
$$\text{slack} = 2\text{ns} + 0\text{ns} - 3\text{ns}$$
$$\text{slack} = -1\text{ns}$$

The negative slack requires a multicycle assignment with a value of 2 and a hold multicycle assignment with a value of 1 to correct. With these assignments from reg_a to reg_b, the setup relationship is then 12 ns, resulting in a slack of 9 ns.

Clock Latency Scenario

Treat the PLL compensation value of –2 ns as latency with a clock skew of 2 ns. Because launch and latch edges are evaluated before latencies are applied, the setup relationship is 10 ns (the period of clk and the PLL) (Figure 7–9).

Figure 7–9. Setup Relationship Using Latency



Equation 2 shows how to calculate the slack value for the path from reg_a to reg_b.

(2)
$$\text{slack} = \text{clock arrival} - \text{data arrival}$$
$$\text{slack} = \text{setup relationship} + \text{clock skew} - \text{reg_to_reg delay}$$
$$\text{slack} = 10\text{ns} + 2\text{ns} - 3\text{ns}$$
$$\text{slack} = 9\text{ns}$$

The slack of 9 ns is identical to the slack computed in the previous example, but because this example uses latency instead of offset, no multicycle assignment is required.

Clock Uncertainty

The Quartus II Classic Timing Analyzer ignores **Clock Setup Uncertainty** and **Clock Hold Uncertainty** assignments when you specify a setup or hold relationship between two clocks. However, the Quartus II TimeQuest Timing Analyzer does not ignore clock uncertainty when you specify a setup or hold relationship between two clocks. Figures 7–10 and 7–11 illustrate the different behavior between the Quartus II TimeQuest and Quartus II Classic Timing Analyzers.

In both figures, the constraints are identical. There is a 20-ns period for `clk_a` and `clk_b`. There is a setup relationship (a **set_max_delay** exception in the Quartus II TimeQuest Timing Analyzer) of 7 ns from `clk_a` to `clk_b`, and a clock setup uncertainty constraint of 1 ns from `clk_a` to `clk_b`. The actual setup relationship in the Quartus II TimeQuest Timing Analyzer is 1 ns less than in the Quartus II Classic Timing Analyzer because of the way they analyze clock uncertainty.

Figure 7–10. Quartus II Classic Timing Analyzer Behavior

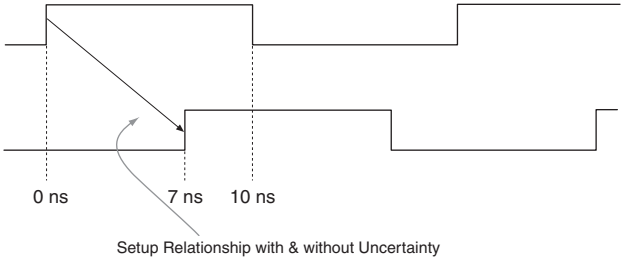
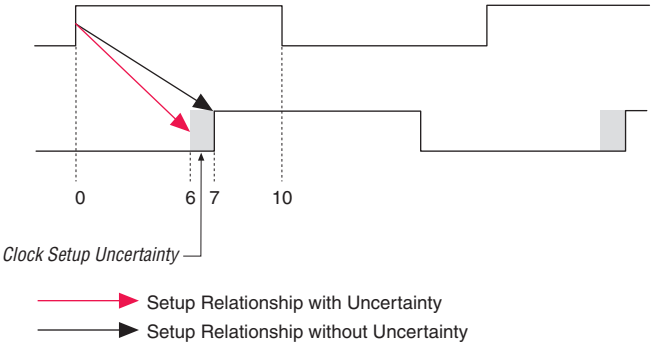


Figure 7–11. Quartus II TimeQuest Timing Analyzer Behavior



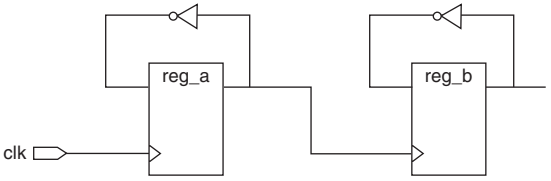
Derived and Generated Clocks

Generated clocks in the Quartus II TimeQuest Timing Analyzer are different than derived clocks in the Quartus II Classic Timing Analyzer. In the Quartus II Classic Timing Analyzer, the source of a derived clock must be a base clock. However, in the Quartus II TimeQuest Timing Analyzer, the source of a generated clock can be any other clock in the design (including virtual clocks), or any node to which a clock propagates through the clock network. Because generated clocks are related through the clock network, you can specify generated clocks for isolated modules, such as IP, without knowing the details of the clocks outside of the module.

In the Quartus II TimeQuest Timing Analyzer, you can specify generated clocks relative to specific edges and edge shifts of a master clock, a feature that the Quartus II Classic Timing Analyzer does not support.

Figure 7–12 shows a simple ripple clock that you should constrain with generated clocks in the Quartus II TimeQuest Timing Analyzer.

Figure 7–12. Generated Clocks Circuit



The Quartus II TimeQuest Timing Analyzer constraints shown in Example 7–4 constrain the clocks in the circuit above. Note that the source of each generated clock can be the input pin of the register itself, not the name of another clock.

Example 7–4. Generated Clock Constraints

```
create_clock -period 10 -name clk clk
create_generated_clock -divide_by 2 -source reg_a|CLK -name reg_a reg_a
create_generated_clock -divide_by 2 -source reg_b|CLK -name reg_b reg_b
```

Automatic Clock Detection

The Quartus II Classic and Quartus II TimeQuest Timing Analyzers identify clock sources of registers that do not have a defined clock source. The Quartus II Classic Timing Analyzer traces back along the clock

network, through registers and logic, until it reaches a top-level port in your design. The Quartus II TimeQuest Timing Analyzer also traces back along the clock network, but it stops at registers.

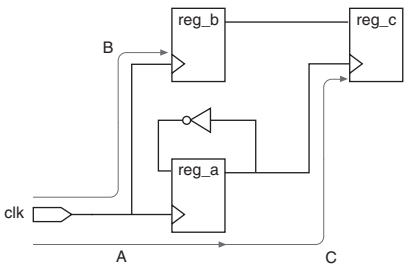
You can use two SDC commands in the Quartus II TimeQuest Timing Analyzer to automatically detect and create clocks for unconstrained clock sources:

- **derive_clocks**—creates clocks on sources of clock pins that do not already have at least one clock sourcing the clock pin
- **derive_pll_clocks**—identifies PLLs and creates generated clocks on the clock output pins

derive_clocks Command

Figure 7–13 shows a simple circuit with a divide-by-2 register and indicates the clock network delays as A, B, and C. The following explanation describes how the Quartus II Classic and Quartus II TimeQuest Timing Analyzers detect the clocks in Figure 7–13.

Figure 7–13. Circuit for derive_clocks Example



The Quartus II Classic Timing Analyzer detects that `clk` is the clock source for registers `reg_a`, `reg_b`, and `reg_c`. It detects that `clk` is the clock source for `reg_c` because it traces back along the clock network for `reg_c` through `reg_a`, until it finds the `clk` port. The Quartus II Classic Timing Analyzer computes the clock arrival time for `reg_c` as $A + C$.

The **derive_clocks** command in the Quartus II TimeQuest Timing Analyzer creates two base clocks, one on the `clk` port and one on `reg_a`, because the command does not trace through registers on the clock network. The clock arrival time for `reg_c` is C because the clock starts at `reg_a`.

To make theQuartus II TimeQuest Timing Analyzer compute the same clock arrival time (*A* + *C*) as the Quartus II Classic Timing Analyzer for **reg_c**, make the following modifications to the clock constraints created by the **derive_clocks** command:

- Change the base clock named `reg_a` to a generated clock
- Make the source the clock pin of `reg_a` (`reg_a|clk`) or the port `clk`
- Add a `-divide_by 2` option

These modifications cause the clock arrival times to `reg_c` to match between the Quartus II Classic Timing Analyzer and the Quartus II TimeQuest Timing Analyzer. However, the clock for `reg_c` is shown as `reg_a` instead of clk, and the launch and latch edges may change for some paths due to the divide-by-2.

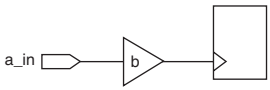
You can use the **derive_clocks** command at the beginning of your design cycle when you do not know all of the clock constraints for your design, but you should not use it during timing sign-off. Instead, you should constrain each clock in your design with the **create_clock** or **create_generated_clocks** commands.

The **derive_clocks** command detects clocks in your design using the following rules:

1. An input clock port is detected as a clock only if there are no other clocks feeding the destination registers.
    a. Input clock ports are not detected automatically if they feed only other base clocks.
    b. If other clocks feed the port's register destinations, the port is assumed to be an enable or data port for a gated clock.
    c. When no clocks are defined, and multiple clocks feed a destination register, the auto-detected clock is selected arbitrarily.
2. All ripple clocks (registers in a clock path) are detected as clocks automatically using the same rules for input clock ports. If both an input port and a register feed register clock pins, the input port is selected as the clock.

The following examples show how the **derive_clocks** command detects clocks in the simple circuit, shown in Figure 7–14.

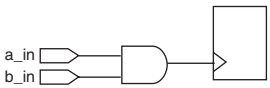
Figure 7–14. Simple Circuit 1



- If you do not make any clock settings, and then you run **derive_clocks**, it detects `a_in` as a clock according to rule 1, because there are no other clocks feeding the register.
- If you create a clock with `b` as its target, and then you run **derive_clocks**, it does not detect `a_in` as a clock according to rule 1a, because `a_in` feeds only another clock.

The following examples show how the **derive_clocks** command detects clocks in the simple circuit shown in Figure 7–15.

Figure 7–15. Simple Circuit 2



- If you do not make any clock settings and then you run **derive_clocks**, it selects a clock arbitrarily, according to rule 1c.
- If you create a clock with `a_in` as its target and then you run **derive_clocks**, it does not detect `b_in` as a clock according to rule 1b, because another clock (`a_in`) feeds the register.

derive_pll_clocks Command

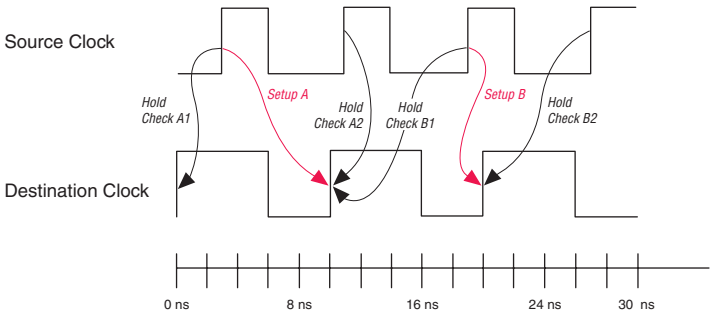
The **derive_pll_clocks** command names the generated clocks according to the names of the PLL output pins by default, and you cannot change these generated clock names. If you want to use your own clock names, you must use the **create_generated_clock** command for each PLL output clock and specify the names with the `-name` option.

If you use the PLL clock-switchover feature, the **derive_pll_clocks** command creates additional generated clocks on each output clock pin of the PLL based on the secondary clock input to the PLL. This may require **set_clock_groups** or **set_false_path** commands to cut the primary and secondary clock outputs. For information about how to make clocks unrelated, refer to "Related and Unrelated Clocks" on page 7–13.

Hold Relationship

The Quartus II TimeQuest and Quartus II Classic Timing Analyzers choose the worst-case hold relationship differently. Refer to Figure 7–16 for sample waveforms to illustrate the different effects.

Figure 7–16. Worst-Case Hold



The Quartus II Classic Timing Analyzer first identifies the worst-case setup relationship. The worst-case setup relationship is **Setup B**. Then the Quartus II Classic Timing Analyzer chooses the worst-case hold relationship (Hold Check B1 or Hold Check B2) for that specific setup relationship, Setup B. The Quartus II Classic Timing Analyzer chooses Hold Check B2 for the worst-case hold relationship.

However, the Quartus II TimeQuest Timing Analyzer calculates worst-case hold relationships for all possible setup relationships and chooses the absolute worst-case hold relationship. The Quartus II TimeQuest Timing Analyzer checks two hold relationships for every setup relationship:

- Data launched by the current launch edge not captured by the previous latch edge (Hold Check A1 and Hold Check B1)
- Data launched by the next launch edge not captured by the current latch edge (Hold Check A2 and Hold Check B2)

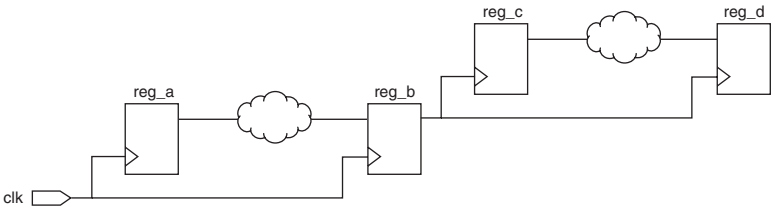
The Quartus II TimeQuest Timing Analyzer chooses Hold Check A2 as the absolute worst-case hold relationship.

Clock Objects

The Quartus II Classic Timing Analyzer treats nodes with clock settings assigned to them as special objects in the timing netlist. Any node in the timing netlist with a clock setting is treated as a clock object, regardless of its actual type, such as a register. When a register has a clock setting

assigned to it, the Quartus II Classic Timing Analyzer does not analyze register-to-register paths beginning or ending at that register. Figure 7–17 shows a circuit that illustrates this situation.

Figure 7–17. Clock Objects



With no clock assignments on any of the registers, the Quartus II Classic Timing Analyzer analyzes timing on the path from `reg_a` to `reg_b`, and from `reg_c` to `reg_d`. If you make a clock setting assignment to `reg_b`, `reg_b` is no longer considered a register node in the netlist, and the path from `reg_a` to `reg_b` is no longer analyzed.

In the Quartus II TimeQuest Timing Analyzer, clocks are abstract objects that are associated with nodes in the timing netlist. The Quartus II TimeQuest Timing Analyzer analyzes the path from `reg_a` to `reg_b` even if there is a clock assigned to `reg_b`.

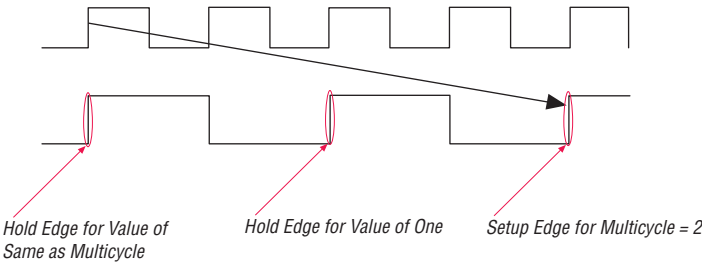
Hold Multicycle

The hold multicycle value numbering scheme is different in the Quartus II Classic and Quartus II TimeQuest Timing Analyzers. Also, you can choose between two values for the default hold multicycle value in the Quartus II Classic Timing Analyzer but you cannot change the default value in the Quartus II TimeQuest Timing Analyzer. The hold multicycle value specifies which clock edge is used for hold analysis when you change the latch edge with a multicycle assignment.

In the Quartus II Classic Timing Analyzer, the hold multicycle value is based on 1, and is the number of clock cycles away from the setup edge. In the Quartus II TimeQuest Timing Analyzer, the hold multicycle value is based on zero, and is the number of clock cycles away from the default hold edge. In the Quartus II TimeQuest Timing Analyzer, the default hold edge is one edge before or after the setup edges. Subtract 1 from any hold multicycle value in theQuartus II Classic Timing Analyzer to compute the equivalent value for the Quartus II TimeQuest Timing Analyzer.

In the Quartus II Classic Timing Analyzer, you can set the default value of the hold multicycle assignment to **One** or **Same as Multicycle**. The default value applies to any multicycle assignment in your design that does not also have a multicycle hold assignment. Figure 7–18 illustrates the difference between **One** and **Same as Multicycle** for a multicycle assignment of 2 using the Quartus II Classic Timing Analyzer.

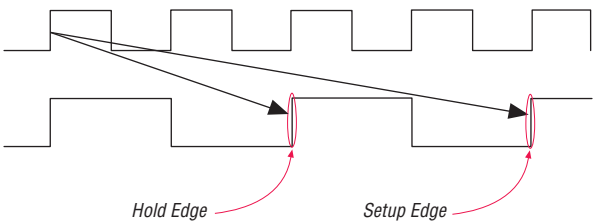
Figure 7–18. Difference Between One and Same As Multicycle



If the default value is **One**, the Quartus II Classic Timing Analyzer uses the clock edge one before the setup edge for hold analysis. If the default value is **Same as Multicycle**, the Quartus II Classic Timing Analyzer uses the clock edge that is <*value of multicycle assignment*> edges back from the setup edge.

Figure 7–19 shows simple waveforms for a cross-clock domain transfer with the indicated setup and hold edges.

Figure 7–19. First Relationship Example



In the Quartus II TimeQuest Timing Analyzer, only a multicycle exception of 2 is required to constrain the design for the indicated setup and hold relationships.

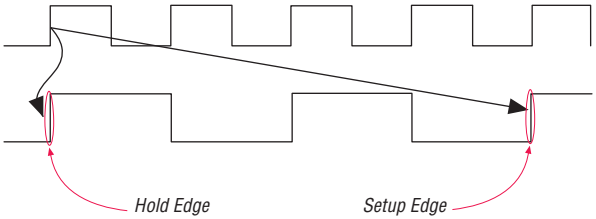
In the Quartus II Classic Timing Analyzer, if the **Default Hold Multicycle** value is **One**, only a multicycle assignment of 2 is required to constrain the design.

In the Quartus II Classic Timing Analyzer, if the **Default Hold Multicycle** value is **Same as Multicycle**, you must make two assignments to constrain the design:

- A multicycle assignment of 2
- A hold multicycle assignment of 1 to override the default value

Figure 7–20 shows simple waveforms for a different cross-clock domain transfer with indicated setup and hold edges. The following explanation shows what exceptions to apply to achieve the desired setup and hold relationships.

Figure 7–20. Second Relationship Example



In the Quartus II TimeQuest Timing Analyzer, you must use the following two exceptions:

- A multicycle exception of 2
- A hold multicycle exception of 1, because the hold edge is one edge behind the default hold edge, which is one edge after the setup edge.

In the Quartus II Classic Timing Analyzer, if the **Default Hold Multicycle** value is **One**, you must make two assignments to constrain the design:

- A multicycle assignment of 2
- A hold multicycle assignment of 2 to override the default value

In the Quartus II Classic Timing Analyzer, if the **Default Hold Multicycle** value is **Same as Multicycle**, only a multicycle assignment of 2 is required to constrain the design.



You should always add a hold multicycle assignment for every multicycle assignment to ensure the correct exceptions are applied regardless of the timing analyzer you use, or, for the Quartus II Classic Timing Analyzer, the **Default Hold Multicycle** setting.

Fitter Behavior

The behavior for one value of the **Optimize hold time** Fitter assignment differs between the Quartus II TimeQuest Timing Analyzer and the Quartus II Classic Timing Analyzer. When you set the Quartus II TimeQuest Timing Analyzer as the default timing analyzer, the **I/O Paths and Minimum TPD Paths** value directs the Fitter to optimize all hold time paths, which has the same affect as the **All Paths** value.

Fitter Performance

If you use the Quartus II TimeQuest Timing Analyzer as your default timing analyzer, the Fitter memory use and compilation time may increase. However, the timing analysis time may decrease.

Reporting

The Quartus II TimeQuest Timing Analyzer provides a more flexible and powerful interface for reporting timing analysis results than the Quartus II Classic Timing Analyzer. Although the interface and constraints are more flexible and powerful, both analyzers use the same device timing models, except for device families that support rise/fall analysis. The Quartus II Classic Timing Analyzer does not support rise/fall analysis, but the Quartus II TimeQuest Timing Analyzer does. Therefore, you may see slightly different delays on identical paths in device families that support rise/fall analysis if you analyze timing in both analyzers.

This means that both analyzers report identical delays along identically constrained paths in your design. The Quartus II TimeQuest Timing Analyzer allows you to constrain some paths that you could not constrain with the Quartus II Classic Timing Analyzer. Differently constrained paths result in different reported values, but for identical paths in your design that are constrained the same way, the delays are exactly the same. Both timing analyzers use the same timing models.



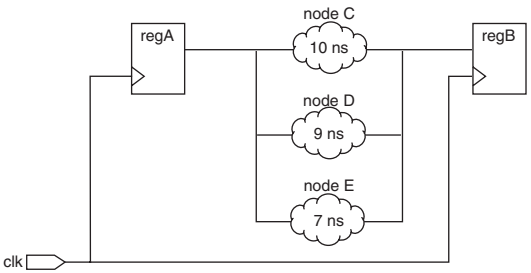
For information about reporting with the Quartus II TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Paths and Pairs

In reporting, the most significant difference between the two analyzers is that the Quartus II TimeQuest Timing Analyzer reports paths, while the Quartus II Classic Timing Analyzer reports pairs. Path reporting means that the analyzer separately reports every path between two registers. Pair reporting means that the analyzer reports only the worst-case path between two registers. One benefit of path reporting over pair reporting is that you can more easily identify common points in failing paths that may be good targets for optimization.

If your design does not meet timing constraints, this reporting difference can give the impression that there are many more timing failures when you use the Quartus II TimeQuest Timing Analyzer. Figure 7–21 shows a sample circuit followed by a description of the differences between path and pair reporting.

Figure 7–21. Failing Paths



There is an 8-ns period constraint on `clk`, resulting in two paths that fail timing: `regA` → `C` → `regB` and `regA` → `D` → `regB`. The Quartus II Classic Timing Analyzer reports only worst-case path `regA` → `C`→ `regB`. The Quartus II TimeQuest Timing Analyzer reports both failing paths `regA` → `C` → `regB` and `regA` → `D` → `regB`. It also reports path `regA` → `E` → `regB` with positive slack.

Default Reports

The Quartus II TimeQuest Timing Analyzer generates only a small number of reports by default, as compared to the Quartus II Classic Timing Analyzer, which generates every report by default. With the Quartus II TimeQuest Timing Analyzer, you generate desired reports on demand.



To learn how to create custom reports, refer to the *Quartus II Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Netlist Names

The Quartus II Classic Timing Analyzer uses register names in reporting, but theQuartus II TimeQuest Timing Analyzer uses register pin names (with the exception of port names of the top-level module). Buried nodes or register names are used when necessary.

Example 7–5 shows how register names are used in Quartus II Classic Timing Analyzer reports.

Example 7–5. Netlist Names in the Quartus II Classic Timing Analyzer

```
Info: + Shortest register to register delay is 0.538 ns
    Info: 1: + IC(0.000 ns) + CELL(0.000 ns) = 0.000 ns; Loc. =
LCFF_X1_Y5_N1;
Fanout = 1; REG Node = 'inst'
    Info: 2: + IC(0.305 ns) + CELL(0.149 ns) = 0.454 ns; Loc. =
LCCOMB_X1_Y5_N20; Fanout = 1; COMB Node = 'inst3~feeder'
    Info: 3: + IC(0.000 ns) + CELL(0.084 ns) = 0.538 ns; Loc. =
LCFF_X1_Y5_N21; Fanout = 1; REG Node = 'inst3'
    Info: Total cell delay = 0.233 ns ( 43.31 % )
    Info: Total interconnect delay = 0.305 ns ( 56.69 % )
```

Example 7–6 shows the same information as presented in a Quartus II TimeQuest Timing Analyzer report. In this example, register pin names are used in place of register names.

Example 7–6. Netlist Names in the Quartus II TimeQuest Timing Analyzer

```
    Info:      3.788      0.250     uTco  inst
    Info:      3.788      0.000 RR  CELL  inst|regout
    Info:      4.093      0.305 RR    IC  inst3~feeder|datad
    Info:      4.242      0.149 RR  CELL  inst3~feeder|combout
    Info:      4.242      0.000 RR    IC  inst3|datain
    Info:      4.326      0.084 RR  CELL  inst3
```

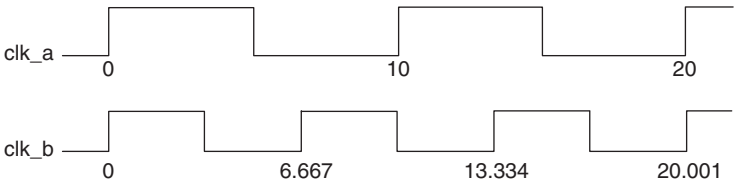
Non-Integer Clock Periods

In some cases when related clock periods are not integer multiples of each other, a lack of precision in clock period definition in the Quartus II TimeQuest Timing Analyzer can result in reported setup or hold relationships of a few picoseconds. In addition, launch and latch times for the relationships can be very large. If you experience this, use the **set_max_delay** and **set_min_delay** exceptions to specify the correct

relationships. The Quartus II Classic Timing Analyzer can maintain additional information about clock frequency that mitigates the lack of precision in clock period definition.

When the clock period cannot be expressed as an integer in terms of picoseconds, then you have the problem detailed in Figure 7–22. This figure shows two clocks: `clk_a` has a 10 ns period, and `clk_b` has a 6.667 ns period.

Figure 7–22. Very Small Setup Relationship



There is a 1 ps setup relationship at 20 ns because you cannot specify the 6.667 ns period beyond picosecond precision. You should apply the maximum and minimum delay exceptions shown in Example 7–7 between the two clocks to specify the correct relationships.

Example 7–7. Minimum and Maximum Delay Exceptions

```
set_max_delay -from [get_clocks clk_a] -to [get_clocks clk_b] 3.333
set_min_delay -from [get_clocks clk_a] -to [get_clocks clk_b] 0
```

Other Features

The Quartus II TimeQuest Timing Analyzer reports time values without units. By default, the units are nanoseconds, and three decimal places are displayed. You can change the default time unit and decimal places with the **set_time_format** command.

When you use the Quartus II TimeQuest Timing Analyzer in a Tcl shell, output is ASCII-formatted, and columns are aligned for easy reading on 80-column consoles. Example 7–8 shows sample output from a **report_timing** command from the Quartus II TimeQuest Timing Analyzer.

Example 7–8. ASCII-Formatted Quartus II TimeQuest Timing Analyzer Report

```
tcl> report_timing -from inst -to inst5
Info: Report Timing: Found 1 setup paths (0 violated).  Worst case slack is 3.634
    Info: -from [get_keepers inst]
    Info: -to [get_keepers inst5]
Info: Path #1: Slack is 3.634
    Info: ===================================================================
    Info: From Node    : inst
    Info: To Node      : inst5
    Info: Launch Clock : clk_a
    Info: Latch Clock  : clk_b
    Info:
    Info: Data Arrival Path:
    Info:
    Info: Total (ns)  Incr (ns)     Type  Node
    Info: ==========  ========= ==  ====  ==================================
    Info:      0.000      0.000           launch edge time
    Info:      2.347      2.347  R        clock network delay
    Info:      2.597      0.250     uTco  inst
    Info:      2.597      0.000 RR  CELL  inst|regout
    Info:      3.088      0.491 RR    IC  inst6|datac
    Info:      3.359      0.271 RR  CELL  inst6|combout
    Info:      3.359      0.000 RR    IC  inst5|datain
    Info:      3.443      0.084 RR  CELL  inst5
    Info:
    Info: Data Required Path:
    Info:
    Info: Total (ns)  Incr (ns)     Type  Node
    Info: ==========  ========= ==  ====  ==================================
    Info:      4.000      4.000           latch edge time
    Info:      7.041      3.041  R        clock network delay
    Info:      7.077      0.036     uTsu  inst5
    Info:
    Info: Data Arrival Time  :     3.443
    Info: Data Required Time :     7.077
    Info: Slack              :     3.634
    Info: ===================================================================
    Info:
1 3.634
```

Scripting API

In versions of the Quartus II software earlier than 6.0, the **::quartus::project** Tcl package contained the following SDC-like commands for making timing assignments:

- **create_base_clock**
- **create_relative_clock**
- **get_clocks**
- **set_clock_latency**
- **set_clock_uncertainty**
- **set_input_delay**
- **set_multicycle_assignment**
- **set_output_delay**
- **set_timing_cut_assignment**

These commands are not SDC-compliant. Beginning with version 6.0, these commands are in a new package called **::quartus::timing_assignment**. To ensure backward compatibility with existing Tcl scripts, the **::quartus::timing_assignment** package is loaded by default in the following executables:

- **quartus**
- **quartus_sh**
- **quartus_cdb**
- **quartus_sim**
- **quartus_stp**
- **quartus_tan**

The **::quartus::timing_assignment** package is not loaded by default in the **quartus_sta** executable. The **::quartus::sdc** Tcl package includes SDC-compliant versions of the commands listed above. That package is available only in the **quartus_sta** executable, and it is loaded by default.

Timing Assignment Conversion

This section describes Quartus II Classic QSF timing assignments and their equivalent Quartus II TimeQuest constraints. You can convert many Quartus II Classic timing assignments to SDC constraints. Some Quartus II Classic timing assignments can be converted to two different SDC constraints, and you must understand the intended functionality of the design to make an appropriate conversion. You cannot convert some Quartus II Classic timing assignments because there is no equivalent SDC constraint.

This section includes the following topics, arranged alphabetically:

Clock Enable Multicycle 7–38
Clock Latency 7–34
Clock Settings 7–37
Clock Uncertainty 7–34
Cut Timing Path 7–52
Default Required f_{MAX} Assignment 7–35
Hold Relationship 7–34
Input and Output Delay 7–39
Inverted Clock 7–35
Maximum Clock Arrival Skew 7–53
Maximum Data Arrival Skew 7–53
Maximum Delay 7–52
Minimum Delay 7–52
Minimum t_{CO} Requirement 7–48
Minimum t_{PD} Requirement 7–51
Multicycle 7–37
Not a Clock 7–35
Setup Relationship 7–33
t_{CO} Requirement 7–45
t_{H} Requirement 7–43
t_{PD} Requirement 7–50
t_{SU} Requirement 7–40
Virtual Clock Reference 7–36

Setup Relationship

The **Setup Relationship** assignment overrides the setup relationship between two clocks. By default, the Quartus II Classic Timing Analyzer automatically calculates the setup relationship based on your clock settings. The QSF variable for the **Setup Relationship** assignment is SETUP_RELATIONSHIP. In the Quartus II TimeQuest Timing Analyzer, use the **set_max_delay** command to specify the maximum setup relationship for a path.

The setup relationship value is the time between latch and launch edges before the Quartus II TimeQuest Timing Analyzer accounts for clock latency, source μt_{CO}, or destination μt_{SU}.

Hold Relationship

The **Hold Relationship** assignment overrides the hold relationship between two clocks. By default, the Quartus II Classic Timing Analyzer automatically calculates the hold relationship based on your clock settings. The QSF variable for the **Hold Relationship** assignment is HOLD_RELATIONSHIP. In the Quartus II TimeQuest Timing Analyzer, use the **set_min_delay** command to specify the minimum hold relationship for a path.

Clock Latency

Table 7–1 shows the equivalent SDC constraints for each of these Quartus II Classic assignments.

Table 7–1. Quartus II Classic and SDC Equivalent Constraints

Quartus II Classic Timing Assignment		SDC Constraint
Assignment Name	**QSF Variable**	
Early Clock Latency	EARLY_CLOCK_LATENCY	set_clock_latency -source -late
Late Clock Latency	LATE_CLOCK_LATENCY	set_clock_latency -source -early

For more information about clock latency support in the Quartus II TimeQuest Timing Analyzer, refer to "Clock Latency" on page 7–15.

Clock Uncertainty

This section describes the conversion for the following Quartus II Classic assignments:

- Clock Setup Uncertainty
- Clock Hold Uncertainty

Table 7–2 shows the equivalent SDC constraints for each of these Quartus II Classic assignments.

Table 7–2. Quartus II Classic and SDC Equivalent Constraints

Quartus II Classic Timing Assignment		SDC Constraint
Assignment Name	**QSF Variable**	
Clock Setup Uncertainty	CLOCK_SETUP_UNCERTAINTY	set_clock_uncertainty -setup
Clock Hold Uncertainty	CLOCK_HOLD_UNCERTAINTY	set_clock_uncertainty -hold

Inverted Clock

The Quartus II Classic Timing Analyzer detects inverted clocks automatically when the clock inversion occurs at the input of the LCELL that contains the register specified in the assignment. You must make an **Inverted Clock** assignment in all other situations for Quartus II Classic Timing Analyzer analysis. The QSF variable for the **Inverted Clock** assignment is INVERTED_CLOCK. The Quartus II TimeQuest Timing Analyzer detects inverted clocks automatically, regardless of the type of inversion circuit, in designs that target device families that support unateness: Stratix® II, Cyclone® II, and HardCopy® II. For designs that target any other device family, you must create a generated clock with the `-invert` option on the output of the cell that inverts the clock.

For more information about unateness, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Not a Clock

The **Not a Clock** assignment directs the Quartus II Classic Timing Analyzer that the specified node is not a clock source when it would normally be detected as a clock because of a global f_{MAX} requirement. The QSF variable for the **Not a Clock** assignment is NOT_A_CLOCK. This assignment is not supported in the Quartus II TimeQuest Timing Analyzer and there is no equivalent constraint. Appropriate clock constraints are created in the Quartus II TimeQuest Timing Analyzer only.

Default Required f_{MAX} Assignment

The **Default Required f_{MAX}** assignment allows you to specify an f_{MAX} requirement for the Quartus II Classic Timing Analyzer for all unconstrained clocks in your design. The QSF variable for the **Default Required f_{MAX}** assignment is FMAX_REQUIREMENT. You can use the

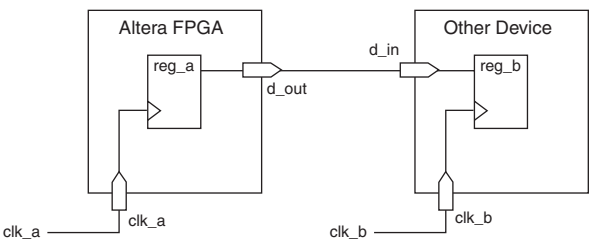
derive_clocks command to create clocks on sources of clock pins in your design that do not already have clocks assigned to them. You should constrain each individual clock in your design with the **create_clock** or **created_generated_clock** command, not the **derive_clocks** command. Refer to "Automatic Clock Detection" on page 7–19 to learn why you should constrain individual clocks in your design.

Virtual Clock Reference

The **Virtual Clock Reference** assignment allows you to define timing characteristics of a reference clock outside the FPGA. The QSF variable for the **Virtual Clock Reference** assignment is VIRTUAL_CLOCK_REFERENCE. The Quartus II TimeQuest Timing Analyzer supports virtual clocks by default, while the Quartus II Classic Timing Analyzer requires the **Virtual Clock Reference** assignment to indicate that a clock setting is for a virtual clock. To create a virtual clock in the Quartus II TimeQuest Timing Analyzer, use the **create_clock** or **create_generated_clock** commands with the `-name` option and no targets.

Figure 7–23 shows a simple circuit that requires a virtual clock, and the following example shows how to constrain the circuit. The circuit shows data transfer between an Altera FPGA and another device, and the clocks for the two devices are not related. You can constrain the path with an output delay assignment, but that assignment requires a virtual clock that defines the clock characteristics of the destination device.

Figure 7–23. Virtual Clock Sample Circuit



Assume the circuit has the following assignments in the Quartus II Classic Timing Analyzer:

- Clock period of 10 ns on `system_clk` (clock for the Altera FPGA)
- Clock period of 8 ns on `virt_clk` (clock for the other device)
- Virtual Clock Reference setting for `virt_clk` is on (indicates that `virt_clk` is a virtual clock)

- Output Maximum Delay of 5 ns on `dataout` with respect to `virt_clk` (constrains the path between the two devices)

The SDC commands shown in Example 7–9 constrain the circuit the same way.

Example 7–9. SDC Constraints

```
# Clock for the Altera FPGA
create_clock -period 10 -name system_clk [get_ports system_clk]
# Virtual clock for the other device, with no targets
create_clock -period 8 -name virt_clk
# Constrains the path between the two devices
set_output_delay -clock virt_clk 5 [get_ports dataout]
```

Clock Settings

The Quartus II Classic Timing Analyzer includes a variety of assignments to describe clock settings. These include duty cycle, f_{MAX}, offset, and others. In the Quartus II TimeQuest Timing Analyzer, use the **create_clock** and **create_generated_clock** commands to constrain clocks.

Multicycle

Table 7–3 shows the equivalent SDC exceptions for each of these Quartus II Classic Timing Analyzer timing assignments.

Table 7–3. Quartus II Classic and SDC Equivalent Exceptions

Quartus II Classic Timing Assignment		SDC Exception
Assignment Name	**QSF Variable**	
Multicycle *(1)*	MULTICYCLE	set_multicycle_path -setup -end
Source Multicycle *(2)*	SRC_MULTICYCLE	set_multicycle_path -setup -start
Multicycle Hold *(3)*	HOLD_MULTICYCLE	set_multicycle_path -hold -end
Source Multicycle Hold	SRC_HOLD_MULTICYCLE	set_multicycle_path -hold -start

Notes to Table 7–3:
(1) A multicycle assignment is also known as a "destination multicycle setup" assignment.
(2) A source multicycle assignment is also known as a "source multicycle setup" assignment.
(3) A multicycle hold is also known as a "destination multicycle hold "assignment.

The default value and numbering scheme for the hold multicycle value is different in the Quartus II Classic and Quartus II TimeQuest Timing Analyzers. Refer to "Hold Multicycle" on page 7–24 for more information

about the difference between the default value and numbering scheme for the hold multicycle value in the Quartus II Classic and Quartus II TimeQuest Timing Analyzers.



For more information about how to convert the hold multicycle value, see the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Clock Enable Multicycle

The Quartus II Classic Timing Analyzer supports the following clock enable multicycle assignments. Corresponding types of multicycle assignments are applied to all registers enabled by the targets of the specified clock.

- Clock Enable Multicycle
- Clock Enable Source Multicycle
- Clock Enable Multicycle Hold
- Clock Enable Source Multicycle Hold

The Quartus II TimeQuest Timing Analyzer supports clock-enabled multicycle constraints with the **get_fanouts** command. Use the **get_fanouts** command to create a collection of nodes that have a common source signal, such as a clock enable.

I/O Constraints

FPGA I/O timing assignments have typically been made with FPGA-centric t_{SU} and t_{CO} requirements for the Quartus II Classic Timing Analyzer. However, the Quartus II Classic Timing Analyzer also supports input and output delay assignments to accommodate industry-standard, system-centric timing constraints. Where possible, you should use system-centric constraints to constrain your designs for the Quartus II TimeQuest Timing Analyzer. Table 7–4 includes Quartus II Classic I/O assignments, the equivalent FPGA-centric SDC constraints, and recommended system-centric SDC constraints.

For setup checks (t_{SU} and t_{CO}), <*latch* – *launch*> equals the clock period for same-clock transfers. For hold checks (t_H and Minimum t_{CO}), <*latch* – *launch*> equals 0 for same clock transfers. Conversions from Quartus II Classic assignments to `set_input_delay` and `set_output_delay` constraints work well only when the source and destination registers' clocks are the same (same clock and polarity). If the source and

destination registers' clocks are different, the conversion may not be straightforward and you should take extra care when converting to `set_input_delay` and `set_output_delay` constraints.

Table 7–4. Quartus II Classic and Quartus II TimeQuest Timing Analyzers Equivalent I/O Constraints

Classic	FPGA-centric SDC	System-centric SDC
t_{SU} Requirement	set_max_delay <*t_{SU} requirement*>	set_input_delay -max <*latch – launch – t_{SU} requirement*>
t_H Requirement	set_min_delay – <*t_H requirement*> *(1)*	set_input_delay -min <*latch – launch + t_H requirement*>
t_{CO} Requirement	set_max_delay <*t_{CO} requirement*>	set_output_delay -max <*latch – launch – t_{CO} requirement*>
Minimum t_{CO} Requirement	set_min_delay <*minimum t_{CO} requirement*>	set_output_delay -min <*latch – launch – minimum t_{CO} requirement*>
t_{PD} Requirement	set_max_delay <*t_{PD} requirement*>	*(2)*
Minimum t_{PD} Requirement	set_min_delay <*minimum t_{PD} requirement*>	*(2)*

Notes to Table 7–4:

(1) Refer to "t_H Requirement" on page 7–43 for an explanation about why this exception uses the negative t_H requirement.

(2) The input and output delays can be used for t_{PD} paths, such that they will be analyzed as a system f_{MAX} path. This is a feature unique to the Quartus II TimeQuest Timing Analyzer.

Input and Output Delay

Table 7–5 shows the equivalent SDC exceptions for each of these Quartus II Classic Timing Analyzer timing assignments.

Table 7–5. Quartus II Classic and SDC Equivalent Exceptions

Quartus II Classic Timing Assignment		SDC Exception
Assignment Name	QSF Variable	
Input Maximum Delay	INPUT_MAX_DELAY	set_input_delay -max
Input Minimum Delay	INPUT_MIN_DELAY	set_input_delay -min
Output Maximum Delay	OUTPUT_MAX_DELAY	set_output_delay -max
Output Minimum Delay	OUTPUT_MIN_DELAY	set_output_delay -min

In some circumstances, you may receive the following warning message when you update the SDC netlist:

```
Warning: For set_input_delay/set_output_delay, port
"<port>" does not have delay for flag (<rise|fall>,
<min|max>)
```

This warning occurs whenever port constraints have maximum or minimum delay assignments, but not both. In the Quartus II Classic Timing Analyzer, device inputs can have **Input Maximum Delay** assignments, **Input Minimum Delay** assignments, or both, and device outputs can have **Output Maximum Delay** assignments, **Output Minimum Delay** assignments, or both.

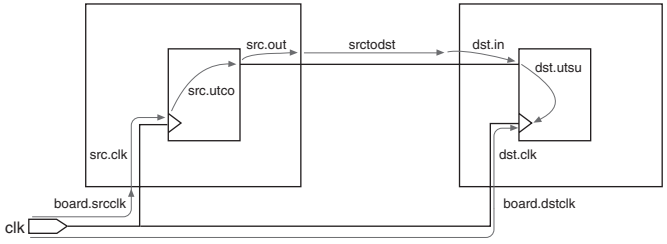
To avoid receiving the warning, your SDC file must specify both the `-max` and `-min` options for each port, or specify neither. If a device I/O in your design includes both the maximum and minimum delay assignments in the Quartus II Classic Timing Analyzer, the conversion utility converts both, and no warning appears about that device I/O. If a device I/O has only maximum or minimum delay assignments in the Quartus II Classic Timing Analyzer, you have the following options:

- Add the missing minimum or maximum delay assignment to the device I/O before performing the conversion.
- Modify the SDC constraint after the conversion to add appropriate `-max` or `-min` values.
- Modify the SDC constraint to remove the `-max` or `-min` option so the value is used for both by default.

t_{SU} Requirement

The **t_{SU} Requirement** assignment specifies the maximum acceptable clock setup time for the input (data) pin. The QSF variable for the **t_{SU} Requirement** assignment is TSU_REQUIREMENT. You can convert the **t_{SU} Requirement** assignment to the **set_max_delay** command or the **set_input_delay** command with the `-max` option. The delay value for the **set_input_delay** command is <*latch* − *launch* − t_{SU} *requirement*>. Refer to the labeled paths in Figure 7–24 to understand the names in Equations 3 and 4.

Figure 7–24. Path Names



Equation 3 shows the derivation of this conversion.

(3)

$$\text{required} - \text{arrival} > 0$$
$$\text{required} = \text{latch} + \text{board.dstclk} + \text{dst.clk} - \text{dst.utsu}$$
$$\text{arrival} = \text{launch} + \text{board.srcclk} + \text{src.clk} + \text{src.utco} + \text{src.out} + \text{srctodst} + \text{dst.in}$$

$$\text{input_delay} = \text{board.srcclk} + \text{src.clk} + \text{src.utcu} + \text{src.out} + \text{srctodst} - \text{board.dstclk}$$

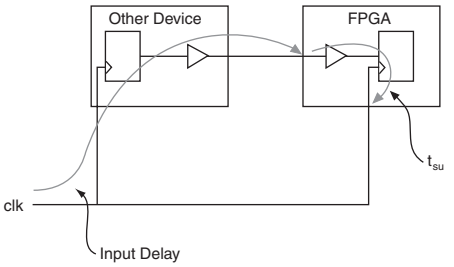
$$\text{required} = \text{latch} + \text{dst.clk} - \text{dst.utsu}$$
$$\text{arrival} = \text{launch} + \text{input_delay} + \text{dst.in}$$
$$(\text{latch} + \text{dst.clk} - \text{dst.utsu}) - (\text{launch} + \text{input_delay} + \text{dst.in}) > 0$$

$$t_{su}\ \text{requirement} - \text{actual}\ t_{su} > 0$$
$$\text{actual}\ t_{su} = \text{dst.in} + \text{dst.utsu} - \text{dst.clk}$$
$$t_{su}\ \text{requirement} - (\text{dst.in} + \text{dst.utsu} - \text{dst.clk}) > 0$$

$$t_{su}\ \text{requirement} = \text{latch} - \text{launch} - \text{input_delay}$$
$$\text{input_delay} = \text{latch} - \text{launch} - t_{su}\text{requirement}$$

The delay value is the difference between the period of the clock source of the register and the **t_{SU} Requirement** value, as shown in Figure 7–25.

Figure 7–25. t_{SU} Requirement



The delay value for the **set_max_delay** command is the **t_{SU} Requirement** value. Equation 4 shows the derivation of this conversion.

(4)

$$\text{required} - \text{arrival} > 0$$
$$\text{required} = \text{latch} + \text{board.dstclk} + \text{dst.clk} - \text{dst.utsu}$$
$$\text{arrival} = \text{launch} + \text{board.srcclk} + \text{src.clk} + \text{src.utco} + \text{src.out} + \text{srctodst} + \text{dst.in}$$
$$\text{max_delay} = \text{latch} + \text{board.dstclk} + - \text{launch} - \text{board.srcclk} - \text{src.clk} - \text{src.out} - \text{srctodst}$$

$$\text{required} = \text{max_delay} + \text{dst.clk} - \text{dst.utsu}$$
$$\text{arrival} = \text{dst.in}$$
$$(\text{max_delay} + \text{dst.clk} - \text{dst.utsu}) - (\text{dst.in}) > 0$$

$$t_{su}\ \text{requirement} - t_{su} > 0$$
$$\text{actual}\ t_{su} = \text{dst.in} + \text{dst.utsu} - \text{dst.clk}$$
$$t_{su}\ \text{requirement} - (\text{dst.in} + \text{dst.utsu} - \text{dst.clk}) > 0$$

$$\text{set_max_delay} = t_{su}\ \text{requirement}$$

Table 7–6 shows the different ways you can make t_{SU} assignments in the Quartus II Classic Timing Analyzer, and the corresponding options for the **set_max_delay** exception.

Table 7–6. t_{SU} Requirement and set_max_delay Equivalence

t_{SU} Requirement Options	set_max_delay Options
-to <*pin*>	-from [get_ports <*pin*>] -to [get_registers *]
-to <*clock*>	-from [get_ports *] -to [get_clocks <*clock*>]
-to <*register*>	-from [get_ports *] -to [get_registers <*register*>]
-from <*pin*> -to <*register*>	-from [get_ports <*pin*>] -to [get_registers <*register*>]
-from <*clock*> -to <*pin*>	-from [get_ports <*pin*>] -to [get_clocks <*clock*>] *(1)*

Notes to Table 7–6:

(1) If the pin in this assignment feeds registers clocked by the same clock, it is equivalent to the first option, `-to` <*pin*>. If the pin feeds registers clocked by different clocks, use **set_input_delay** to constrain the paths properly.

To convert a global t_{SU} assignment to an equivalent SDC exception, use the command shown in Example 7–10.

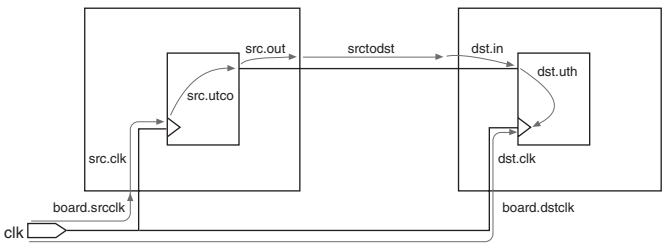
Example 7–10. Converting a Global t_{SU} Assignment to an Equivalent SDC Exception

```
set_max_delay -from [all_inputs] -to [all_registers] <tSU value>
```

t_H Requirement

The **t_H Requirement** specifies the maximum acceptable clock hold time for the input (data) pin. The QSF variable for the **t_H Requirement** assignment is TH_REQUIREMENT. You can convert the **t_H Requirement** assignment to the **set_min_delay** command, or the **set_input_delay** command with the `-min` option. The delay value for the **set_input_delay** command is <*latch* − *launch* + t_H *requirement*>. Refer to the labeled paths in Figure 7–26 to understand the names in Equations 5 and 6.

Figure 7–26. Path Names



Equation 5 shows the derivation of this calculation.

(5)

$$\text{arrival} - \text{required} > 0$$
$$\text{arrival} = \text{launch} + \text{board.srcclk} + \text{src.clk} + \text{src.utco} + \text{src.out} + \text{srctodst} + \text{dst.in}$$
$$\text{required} = \text{latch} + \text{board.dstclk} + \text{dst.clk} + \text{dst.uth}$$
$$\text{input_delay} = \text{board.srcclk} + \text{src.clk} + \text{srcutcu} + \text{src.out} + \text{srctodst} - \text{board.dstclk}$$

$$\text{arrival} = \text{launch} + \text{input_delay} + \text{dst.in}$$
$$\text{required} = \text{latch} + \text{dst.clk} + \text{dst.uth}$$
$$(\text{launch} + \text{input_delay} + \text{dst.in}) - (\text{latch} + \text{dst.clk} + \text{dst.uth}) > 0$$

$$t_H \text{ requirement} - \text{actual } t_H > 0$$
$$\text{actual } t_H = \text{dst.clk} + \text{dst.uth} - \text{dst.in}$$
$$t_H \text{ requirement} - (\text{dst.clk} + \text{dst.uth} - \text{dst.in}) > 0$$

$$t_H \text{ requirement} = \text{launch} - \text{latch} + \text{input_delay}$$
$$\text{input_delay} = \text{latch} - \text{launch} + t_H \text{ requirement}$$

The delay value for the **set_min_delay** command is the **t_H Requirement** value. Equation 6 shows the derivation of this conversion.

(6)

$$\text{arrival} - \text{required} > 0$$
$$\text{arrival} = \text{dst.in}$$
$$\text{required} = \text{min_delay} + \text{dst.clk} + \text{dst.uth}$$

$$\text{dst.in} - (\text{min_delay} + \text{dst.clk} + \text{dst.uth})$$

$$t_H \text{ requirement} - \text{actual } t_H > 0$$
$$\text{actual } t_H = \text{dst.clk} + \text{dst.uth} - \text{dst.in}$$
$$t_H \text{ requirement} - (\text{dst.clk} + \text{dst.uth} - \text{dst.in}) > 0$$

$$\text{set_min_delay} = -t_H \text{ requirement}$$

Table 7–7 shows the different ways you can make t_H assignments in the Quartus II Classic Timing Analyzer, and the corresponding options for the **set_min_delay** command.

Table 7–7. t_H Requirement and set_min_delay Equivalence

t_H Requirement Options	set_min_delay Options
-to <*pin*>	-from [get_ports <*pin*>] -to [get_registers *]
-to <*clock*>	-from [get_ports *] -to [get_clocks <*clock*>]
-to <*register*>	-from [get_ports *] -to [get_registers <*register*>]
-from <*pin*> -to <*register*>	-from [get_ports <*pin*>] -to [get_registers <*register*>]
-from <*clock*> -to <*pin*>	-from [get_ports <*pin*>] -to [get_clocks <*clock*>] *(1)*

Notes to Table 7–7:

(1) If the pin in this assignment feeds registers clocked by the same clock, it is equivalent to the first option, `-to <pin>`. If the pin feeds registers clocked by different clocks, use **set_input_delay** to constrain the paths properly. Refer to "Input and Output Delay" on page 7–39 for additional information.

To convert a global t_H assignment to an equivalent SDC exception, use the command shown in Example 7–11.

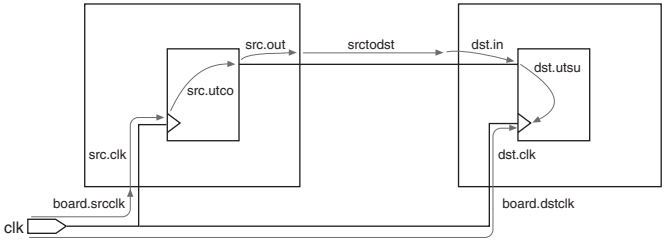
Example 7–11. Converting a Global t_H Assignment to an Equivalent SDC Exception

```
set_min_delay -from [all_inputs] -to [all registers] <negative t_H value>
```

t_{CO} Requirement

The **t_{CO} Requirement** assignment specifies the maximum acceptable clock to output delay to the output pin. The QSF variable for the **t_{CO} Requirement** assignment is TCO_REQUIREMENT. You can convert the **t_{CO} Requirement** assignment to the **set_max_delay** command or the **set_output_delay** with the -max option. The delay value for the **set_output_delay** command is <*latch − launch + t_{CO} requirement*>. Refer to the labeled paths in Figure 7–27 to understand the names in Equations 7 and 8.

Figure 7–27. Path Names



Equation 7 shows the derivation of this conversion.

(7)

$$\text{required} - \text{arrival} > 0$$
$$\text{required} = \text{latch} - \text{output_delay}$$
$$\text{arrival} = \text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$\text{output_delay} = \text{srctodst} + \text{dst.in} + \text{dst.utsu} - \text{dst.clk} - \text{board.dstc.k} + \text{board.srcclk}$$

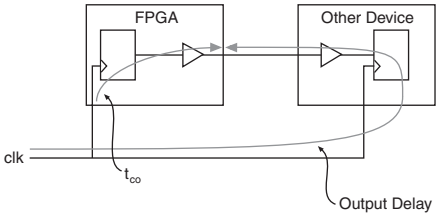
$$(\text{latch} - \text{output_delay}) - (\text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}) > 0$$

$$t_{co} \text{ requirement} - \text{actual } t_{co} > 0$$
$$\text{actual } t_{co} = \text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$t_{co} \text{ requirement} - (\text{src.clk} + \text{src.utco} + \text{src.out}) > 0$$

$$t_{co} \text{ requirement} = \text{latch} - \text{launch} - \text{output_delay}$$
$$\text{output_delay} = \text{latch} - \text{launch} - t_{co} \text{ requirement}$$

The delay value is the difference between the period of the clock source of the register and the **t_{CO} Requirement** value, as illustrated in Figure 7–28.

Figure 7–28. t_{CO} Requirement



The delay value for the **set_max_delay** command is the **t_{CO} Requirement** value. Equation 8 shows the derivation of this conversion.

(8)

$$\text{required} - \text{arrival} > 0$$
$$\text{required} = \text{set_max_delay}$$
$$\text{arrival} = \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$\text{set_max_delay} - (\text{src.clk} + \text{src.utco} + \text{src.out}) > 0$$

$$t_{co}\ \text{requirement} - \text{actual}\ t_{co} > 0$$
$$\text{actual}\ t_{co} = \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$t_{co}\ \text{requirement} - (\text{src.clk} + \text{src.utco} + \text{src.out}) > 0$$

$$\text{set_max_delay} = t_{co}\ \text{requirement}$$

Table 7–8 shows the different ways you can make t_{CO} assignments in the Quartus II Classic Timing Analyzer, and the corresponding options for the **set_max_delay** exception.

Table 7–8. t_{CO} Requirement and set_max_delay Equivalence

t_{CO} Requirement Options	set_max_delay Options
-to <*pin*>	-from [get_registers *] -to [get_ports <*pin*>]
-to <*clock*>	-from [get_clocks <*clock*>] -to [get_ports *]
-to <*register*>	-from [get_registers <*register*>] -to [get_ports *]
-from <*register*> -to <*pin*>	-from [get_registers <*register*>] -to [get_ports <*pin*>]
-from <*clock*> -to <*pin*>	-from [get_clocks <*clock*>] -to [get_ports <*pin*>] *(1)*

Notes to Table 7–8:

(1) If the pin in this assignment feeds registers clocked by the same clock, it is equivalent to the first option, `-to <pin>`. If the pin feeds registers clocked by different clocks, you should use **set_output_delay** to constrain the paths properly.

To convert a global t_{CO} assignment to an equivalent SDC exception, use the command in Example 7–12.

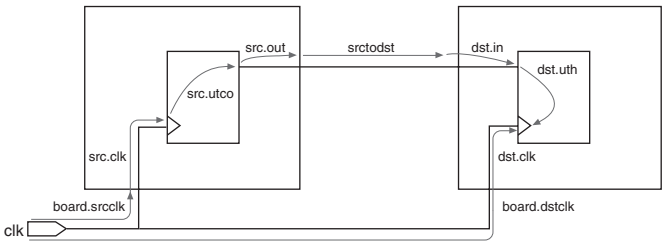
Example 7–12. Converting a Global t_{CO} Assignment to an Equivalent SDC Exception

```
set_max_delay -from [all registers] -to [all_outputs] <tCO value>
```

Minimum t_{CO} Requirement

The **Minimum t_{CO} Requirement** assignment specifies the minimum acceptable clock to output delay to the output pin. The QSF variable for the **Minimum t_{CO} Requirement** assignment is MIN_TCO_REQUIREMENT. You can convert the **Minimum t_{CO} Requirement** assignment to the **set_min_delay** command or the **set_output_delay** command with the `-min` option. The delay value for the **set_output_delay** command is <*latch – launch + minimum t_{CO} requirement*>. Refer to the labeled paths in Figure 7–29 to understand the names in Equations 9 and 10.

Figure 7–29. Path Names



Equation 9 shows the derivation of this conversion.

(9)

$$\text{arrival} + \text{required} > 0$$
$$\text{arrival} = \text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$\text{required} = \text{latch} - \text{output_delay}$$
$$\text{output_delay} = \text{srctodst} + \text{dst.in} - \text{dst.uth} - \text{dst.clk} - \text{board.dstclk} + \text{board.srcclk}$$

$$(\text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}) - (\text{latch} - \text{output_delay}) > 0$$

$$\text{minimum } t_{co} - \text{minimum } t_{co}\text{requirement} > 0$$
$$\text{minimum } t_{co} = \text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$(\text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}) - \text{minimum } t_{co} \text{ requirement} > 0$$

$$\text{minimum } t_{co} \text{ requirement} = \text{latch} - \text{launch} - \text{output_delay}$$
$$\text{output_delay} = \text{latch} - \text{launch} - \text{minimum } t_{co} \text{ requirement}$$

The delay value for the **set_min_delay** command is the **Minimum t_{CO} Requirement**. Equation 10 shows the derivation of this conversion.

(10)

$$\text{arrival} - \text{required} > 0$$
$$\text{arrival} = \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$\text{required} = \text{min_delay}$$
$$(\text{src.clk} + \text{src.utco} + \text{src.out}) - (\text{set_min_delay}) > 0$$

$$\text{minimum } t_{co} - \text{minimum } t_{co} \text{ requirement} > 0$$
$$\text{minimum } t_{co} = \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$(\text{src.clk} + \text{src.utco} + \text{src.out}) - \text{minimum } t_{co} \text{requirement} > 0$$

$$\text{set_min_delay} = \text{minimum } t_{co} \text{ requirement}$$

Table 7–9 shows the different ways you can make minimum t_{CO} assignments in the Quartus II Classic Timing Analyzer, and the corresponding options for the **set_min_delay** exception.

Table 7–9. Minimum t_{CO} Requirement and set_min_delay Equivalence

Minimum t_{CO} Requirement Options	set_min_delay Options
-to <*pin*>	-from [get_registers *] -to [get_ports <*pin*>]
-to <*clock*>	-from [get_clocks <*clock*>] -to [get_ports *]
-to <*register*>	-from [get_registers <*register*>] -to [get_ports *]
-from <*register*> -to <*pin*>	-from [get_registers <*register*>] -to [get_ports <*pin*>]
-from <*clock*> -to <*pin*>	-from [get_clocks <*clock*>] -to [get_ports <*pin*>] *(1)*

Notes to Table 7–9:

(1) If the pin in this assignment feeds registers clocked by the same clock, it is equivalent to the first option, `-to` <pin>. If the pin feeds registers clocked by different clocks, you should use **set_output_delay** to constrain the paths properly.

To convert a global **Minimum t_{CO} Requirement** to an equivalent SDC exception, use the command shown in Example 7–13.

Example 7–13. Converting a Global minimum t_{CO} Requirement to an Equivalent SDC Exception

```
set_min_delay -from [all_registers] -to [all_outputs] <minimum tCO value>
```

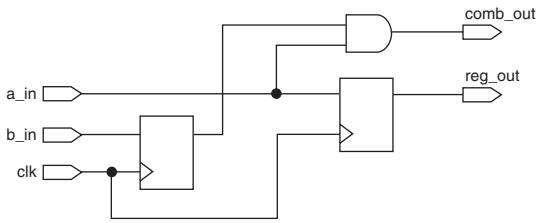
t_{PD} Requirement

The **t_{PD} Requirement** assignment specifies the maximum acceptable input to non-registered output delay, that is, the time required for a signal from an input pin to propagate through combinational logic and appear at an output pin. The QSF variable for the **t_{PD} Requirement** assignment is TPD_REQUIREMENT. You can use the **set_max_delay** command in the Quartus II TimeQuest Timing Analyzer as an equivalent constraint as long as you account for input and output delays. The **t_{PD} Requirement** assignment does not take into account input and output delays, but the **set_max_delay** exception does, so you must modify the **set_max_delay** value to take into account input and output delays.

Combinational Path Delay Scenario

Figure 7–30 shows a simple circuit followed by an example of a **t_{PD} Requirement** to **set_max_delay** conversion.

Figure 7–30. t_{PD} Example



Assume the circuit has the following assignments in the Quartus II Classic Timing Analyzer:

- Clock period of 10 ns
- **t_{PD} Requirement** from `a_in` to `comb_out` of 10 ns
- **Input Max Delay** on `a_in` relative to `clk` of 2 ns
- **Output Max Delay** on `comb_out` relative to `clk` of 2 ns

The path from `a_in` to `comb_out` is not affected by the input and output delays. The slack is equal to the <*t_{PD} Requirement from* `a_in` *to* `comb_out`> – <*path delay from* `a_in` *to* `comb_out`>.

Assume the circuit has the SDC constraints shown in Example 7–14 in the Quartus II TimeQuest Timing Analyzer:

Example 7–14. SDC Constraints

```
create_clock -period 10 –name clk [get_ports clk]
set_max_delay -from a_in -to comb_out 10
set_input_delay -clk clk 2 [get_ports a_in]
set_output_delay –clk clk 2 [get_ports comb_out]
```

The path from a_in to comb_out is affected by the input and output delays. The slack is equal to:

<*set_max_delay value from* a_in *to* comb_out> – <*input delay*> – <*output delay*> – <*path delay from* a_in *to* comb_out>

To convert a global **t_{PD} Requirement** assignment to an equivalent SDC exception, use the command shown in Example 7–15. You should add the input and output delays to the value of your converted **t_{PD} Requirement** (**set_max_delay** exception value) to achieve an equivalent SDC exception.

Example 7–15. Converting a Global t_{PD} Requirement Assignment to an Equivalent SDC Exception

```
set_max_delay -from [all_inputs] -to [all_outputs] <value>
```

Minimum t_{PD} Requirement

The **Minimum t_{PD} Requirement** assignment specifies the minimum acceptable input to non-registered output delay, that is, the minimum time required for a signal from an input pin to propagate through combinational logic and appear at an output pin. The QSF variable for the **Minimum t_{PD} Requirement** assignment is MIN_TPD_REQUIREMENT. You can use the **set_min_delay** command in the Quartus II TimeQuest Timing Analyzer as an equivalent constraint as long as you account for input and output delays. The **Minimum t_{PD} Requirement** assignment does not take into account input and output delays, but the **set_min_delay** exception does.

Refer to "Combinational Path Delay Scenario" on page 7–50 to see how input and output delays affect minimum and maximum delay exceptions.

To convert a global **Minimum t_{PD} Requirement** assignment to an equivalent SDC exception, use the following command:

Example 7–16. Converting a Global Minimum t_{PD} Requirement Assignment to an Equivalent SDC Exception

```
set_min_delay -from [all_inputs] -to [all_outputs] <value>
```

Cut Timing Path

The **Cut Timing Path** assignment in the Quartus II Classic Timing Analyzer is equivalent to the **set_false_path** command in the Quartus II TimeQuest Timing Analyzer. The QSF variable for the **Cut Timing Path** assignment is CUT.

Maximum Delay

The **Maximum Delay** assignment specifies the maximum required delay for the following types of paths:

- Pins to registers
- Registers to registers
- Registers to pins

The QSF variable for the **Maximum Delay** assignment is MAX_DELAY. This requirement overwrites the requirement computed from the clock setup relationship and clock skew. There is no equivalent constraint in the Quartus II TimeQuest Timing Analyzer.



The **Maximum Delay** assignment for the Quartus II Classic Timing Analyzer is not related to the **set_max_delay** exception in the Quartus II TimeQuest Timing Analyzer.

Minimum Delay

The **Minimum Delay** assignment specifies the minimum required delay for the following types of paths:

- Pins to registers
- Registers to registers
- Registers to pins

The QSF variable for the **Minimum Delay** assignment is `MIN_DELAY`. This requirement overwrites the requirement computed from the clock hold relationship and clock skew. There is no equivalent constraint in the Quartus II TimeQuest Timing Analyzer.



The **Minimum Delay** assignment for the Quartus II Classic Timing Analyzer is not related to the **set_min_delay** exception in the Quartus II TimeQuest Timing Analyzer.

Maximum Clock Arrival Skew

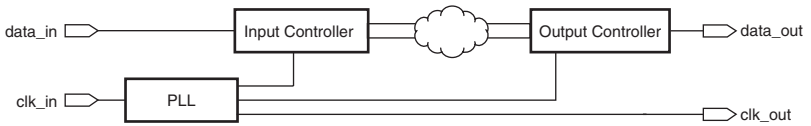
The **Maximum Clock Arrival Skew** assignment specifies the maximum clock skew between a set of registers. The QSF variable for the **Maximum Clock Arrival Skew** assignment is `MAX_CLOCK_ARRIVAL_SKEW`. In the Quartus II Classic Timing Analyzer, this assignment is specified between a clock node name and a set of registers. **Maximum Clock Arrival Skew** is not supported in the Quartus II TimeQuest Timing Analyzer.

Maximum Data Arrival Skew

The **Maximum Data Arrival Skew** assignment specifies the maximum data arrival skew between a set of registers, pins, or both. The QSF variable for the **Maximum Data Arrival Skew** assignment is `MAX_DATA_ARRIVAL_SKEW`. In this case, the data arrival delay represents the t_{CO} from the clock to the given register, pin, or both. This assignment is specified between a clock node and a set of registers, pins, or both.

The Quartus II TimeQuest Timing Analyzer does not support a constraint to specify maximum data arrival skew, but you can specify setup and hold times relative to a clock port to constrain an interface like this. Figure 7–31 shows a simplified source-synchronous interface used in the following example.

Figure 7–31. Source-Synchronous Interface Diagram

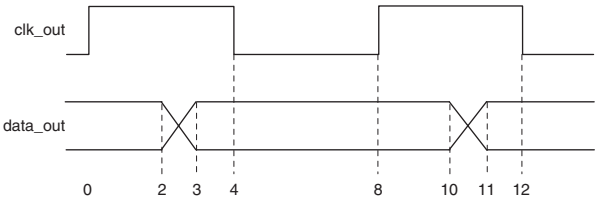


Constraining Skew on an Output Bus

This example constrains the interface so that all bits of the `data_out` bus go off-chip between 2 and 3 ns after the `clk_out` signal. Assume that `clock_in` and `clock_out` have a period of 8 ns.

The following equations and example shows how to create timing requirements that satisfy the timing relationships shown in Figure 7–32.

Figure 7–32. Source-Synchronous Timing Diagram



Equation 11 shows how to compute the value for the **set_output_delay -min** command that creates the 2 ns hold requirement on the destination. For hold requirement calculations in which source and destination clocks are the same, <*latch*> – <*launch*> = 0.

(11)
$$\text{latch} - \text{launch} = 0\,\text{ns}$$
$$\text{output delay} = \text{latch} - \text{launch} - 2\,\text{ns}$$
$$\text{output delay} = -2\,\text{ns}$$

Equation 12 shows how to compute the value for the **set_output_delay** command that creates the 3 ns setup requirement on the destination. For setup requirement calculations in which source and destination clocks are the same, <*latch*> – <*launch*> = clock period.

(12)
$$\text{latch} - \text{launch} = 8\,\text{ns}$$
$$\text{output delay} = \text{latch} - \text{launch} - 3\,\text{ns}$$
$$\text{output delay} = 5\,\text{ns}$$

Refer to "I/O Constraints" on page 7–38 for an explanation of the above equations.

Example 7–17 shows the three constraints together.

Example 7–17. Constraining a DDR Interface

```
set period 8.000
create_clock -period $period \
             -name clock_in \
             clock_in
derive_pll_clocks
set_output_delay -add_delay \
    -clock ddr_pll_1_inst|altpll_component|pll|CLK[0] \
    -reference_pin clk_out \
    -min -2.000 \
    [get_ports data_out*]
set_output_delay -add_delay \
    -clock ddr_pll_1_inst|altpll_component|pll|CLK[0] \
    -reference_pin clk_out \
    -max [expr $period - 3.000] \
    [get_ports data_out*]
```

Conversion Utility

The Quartus II TimeQuest Timing Analyzer includes a conversion utility to help you convert Quartus II Classic timing assignments in a QSF file to SDC constraints in an SDC file. The utility can use information from your project report database (in the **\db** folder), if it exists, so you should compile your design before the conversion.



The conversion utility ignores all disabled QSF assignments. Disabled assignments say **No** in the **Enabled?** column of the Assignment Editor, and include the `-disable` option in the QSF file.

Refer to "Conversion Utility" on page 7–3 to learn how to run the conversion utility.

Unsupported Global Assignments

The conversion utility checks whether any of the global timing assignments in Table 7–10 exist in your project. Any global assignments not supported by the conversion utility are ignored during the conversion. Refer to the indicated page for information about each assignment, and how to manually convert these global assignments to SDC commands.

Table 7–10. Global Timing Assignments

Assignment Name	QSF Variable	More Information
t_{SU} Requirement	TSU_REQUIREMENT	page 7–40
t_{H} Requirement	TH_REQUIREMENT	page 7–43
t_{CO} Requirement	TCO_REQUIREMENT	page 7–45
Minimum t_{CO} Requirement	MIN_TCO_REQUIREMENT	page 7–48
t_{PD} Requirement	TPD_REQUIREMENT	page 7–50
Minimum t_{PD} Requirement	MIN_TPD_REQUIREMENT	page 7–51

Recommended Global Assignments

Once any unsupported assignments have been identified, the conversion utility checks the global assignments in Table 7–11 to ensure they match the specified values.

Table 7–11. Recommended Global Assignments

Quartus II Classic Assignment Name	QSF Variable	Value
Cut off clear and preset signal paths	CUT_OFF_CLEAR_AND_PRESET_PATHS	ON
Cut off feedback from I/O pins	CUT_OFF_IO_PIN_FEEDBACK	ON
Cut off read during write signal paths	CUT_OFF_READ_DURING_WRITE_PATHS	ON
Analyze latches as synchronous elements	ANALYZE_LATCHES_AS_SYNCHRONOUS_ELEMENTS	ON
Enable Clock Latency	ENABLE_CLOCK_LATENCY	ON
Display Entity Name	PROJECT_SHOW_ENTITY_NAME	ON

The following assignments are checked to ensure the functionality of the Quartus II Classic Timing Analyzer with the specified values corresponds to the behavior of the Quartus II TimeQuest Timing Analyzer.

- **Cut off clear and preset signal paths**—Quartus II The TimeQuest Timing Analyzer does not support this functionality. You should use Recovery and Removal analysis instead to analyze register control paths. The Quartus II Classic Timing Analyzer does not support this option.
- **Cut off feedback from I/O pins**—The Quartus II TimeQuest Timing Analyzer does not match the functionality of the Quartus II Classic Timing Analyzer when this assignment is OFF.
- **Cut off read during write signal paths**—The Quartus II TimeQuest Timing Analyzer does not match the functionality of the Quartus II Classic Timing Analyzer when this assignment is OFF.
- **Analyze latches as synchronous elements**—The Quartus II TimeQuest Timing Analyzer analyzes latches as synchronous elements by default and does not match the functionality of the Quartus II Classic Timing Analyzer when this assignment is OFF. Beginning with version 5.1 of the Quartus II software, the Quartus II Classic Timing Analyzer analyzes latches as synchronous elements by default.
- **Enable Clock Latency**—The Quartus II TimeQuest Timing Analyzer includes clock latency in its calculations. The Quartus II TimeQuest Timing Analyzer does not match the functionality of the Quartus II Classic Timing Analyzer when this assignment is OFF. Latency on a clock can be viewed as a simple delay on the clock path, and affects clock skew. This is in contrast to an offset, which alters the setup and hold relationship between two clocks. Refer to "Offset and Latency Example" on page 7–15 for an example of the different effects of offset and latency. When you turn on **Enable Clock Latency** in the Quartus II Classic Timing Analyzer, it affects the following aspects of timing analysis:
  - **Early Clock Latency** and **Late Clock Latency** assignments are honored
  - The compensation delay of a PLL is analyzed as latency
  - For clock settings where you do not specify an offset, the automatically computed offset is treated as latency.
- **Display Entity Name**—Any entity-specific assignments are ignored in the Quartus II TimeQuest Timing Analyzer because they do not include the entity name when this option is ON.

If your design meets timing requirements in the Quartus II Classic Timing Analyzer without all of the settings recommended in Table 7–11 on page 7–56, you should perform one of the following actions.

- Change the settings and re-constrain and re-verify as necessary.
  or
- Add or modify SDC constraints as appropriate because analysis in the Quartus II TimeQuest Timing Analyzer may be different after conversion.

Clock Conversion

Next, the conversion utility adds the **derive_pll_clocks** command to the SDC file. This command creates generated clocks on all PLL outputs in your design each time the SDC file is read. The command does not add a clock on the FPGA port that drives the PLL input.

The conversion utility includes the **derive_pll_clocks -use_tan_name** command in the SDC file it creates. The `-use_tan_name` option overrides the default clock naming behavior (the PLL pin name) so the clock name is the same as the net name in the Quartus II Classic Timing Analyzer.

Including the `-use_tan_name` option ensures that the conversion utility converts clock-to-clock exceptions properly. If you remove the `-use_tan_name` option, you must also edit references to the clock names in other SDC commands so that they match the PLL pin names.

If your design includes a global f_{MAX} assignment, the assignment is converted to a **derive_clocks** command. The behavior of a global f_{MAX} assignment is different from the behavior of clocks created with the **derive_clocks** command, and you should use the **report_clocks** command when you review conversion results to evaluate the clock settings. Refer to "Automatic Clock Detection" on page 7–19 for an explanation of the differences. As soon as you know the appropriate clock settings, you should use **create_clock** or **create_generated_clock** commands instead of the **derive_clocks** command.



The conversion utility adds a **post_message** command before the **derive_clocks** command to remind you that the clocks are derived automatically. The Quartus II TimeQuest Timing Analyzer displays the reminder the first time it reads the SDC file. Remove or comment out the **post_message** command to prevent the message from displaying.

Next, the conversion utility identifies and converts clock settings in the QSF file. If a project database exists, the utility also identifies and converts any additional clocks in the report file that are not in the QSF, such as PLL base clocks.



If you change the PLL input frequency, you must modify the SDC constraint manually.

The conversion utility ignores clock offsets on generated clocks. Refer to "Clock Offset" on page 7–14 for information about how to use offset values in the Quartus II TimeQuest Timing Analyzer.

Instance Assignment Conversion

Next, the conversion utility converts the following instance assignments in Table 7–12. Refer to the indicated page for information about each assignment.

Table 7–12. Instance Timing Assignments

Assignment Name	QSF Variable	More Information
Late Clock Latency	LATE_CLOCK_LATENCY	page 7–34
Early Clock Latency	EARLY_CLOCK_LATENCY	
Clock Setup Uncertainty	CLOCK_SETUP_UNCERTAINTY	page 7–34
Clock Hold Uncertainty	CLOCK_HOLD_UNCERTAINTY	
Multicycle *(1)*	MULTICYCLE	page 7–37
Source Multicycle *(2)*	SRC_MULTICYCLE	
Multicycle Hold *(3)*	HOLD_MULTICYCLE	
Source Multicycle Hold	SRC_HOLD_MULTICYCLE	
Clock Enable Multicycle	CLOCK_ENABLE_MULTICYCLE	page 7–38
Clock Enable Source Multicycle	CLOCK_ENABLE_SOURCE_MULTICYCLE	
Clock Enable Multicycle Hold	CLOCK_ENABLE_MULTICYCLE_HOLD	
Clock Enable Source Multicycle Hold	CLOCK_ENABLE_SOURCE_MULTICYCLE_HOLD	
Cut Timing Path	CUT	page 7–52
Input Maximum Delay	INPUT_MAX_DELAY	page 7–39
Input Minimum Delay	INPUT_MIN_DELAY	
Output Maximum Delay	OUTPUT_MAX_DELAY	
Output Minimum Delay	OUTPUT_MIN_DELAY	

Notes to Table 7–12:
(1) A multicycle assignment can also be known as a "destination multicycle setup" assignment.
(2) A source multicycle assignment can also be known as a "source multicycle setup" assignment.
(3) A multicycle hold can also be known as a "destination multicycle hold" assignment.

Depending on input and output delay assignments, you may receive a warning message when the SDC file is read. The message warns that the **set_input_delay** commands, **set_output_delay** commands, or both are

missing the -max option, -min option, or both. Refer to "Input and Output Delay" on page 7–39 for an explanation of why the warning occurs and how to avoid it.

Beginning in version 7.1 of the Quartus II software, the conversion utility automatically adds multicycle hold exceptions for each multicycle setup assignment. The value of each multicycle hold exception depends on the **Default hold multicycle** assignment value in your project. If the value is **One**, the conversion utility uses a value of 0 (zero) for each multicycle hold exception it adds. If the value is **Same as multicycle**, the conversion utility uses a value one less than the corresponding multicycle setup assignment value for each multicycle hold exception it adds. Refer to "Hold Multicycle" on page 7–24 for more information on hold multicycle differences between the Quartus II Classic and Quartus II TimeQuest Timing Analyzers.

Next, the conversion utility converts the following instance assignments in Table 7–13. Refer to the indicated page for information about each assignment. If the t_{PD} and minimum t_{PD} assignment targets also have input or output delays that apply to them, the t_{PD} and minimum t_{PD} conversion values may be incorrect. This is described in more detail on the indicated pages for the appropriate assignments.

Table 7–13. Instance Timing Assignments

Assignment Name	QSF Variable	More Information
t_{PD} Requirement *(1)*	TPD_REQUIREMENT	page 7–50
Minimum t_{PD} Requirement *(1)*	MIN_TPD_REQUIREMENT	page 7–51
Setup Relationship	SETUP_RELATIONSHIP	page 7–33
Hold Relationship	HOLD_RELATIONSHIP	page 7–34

Note to Table 7–13:

(1) Refer to "t_{PD} and Minimum t_{PD} Requirement Conversion" on page 7–71 for more information about how the conversion utility converts single-point t_{PD} and minimum t_{PD} assignments.

The conversion utility converts Quartus II Classic I/O timing assignments to FPGA-centric SDC constraints. Table 7–14 includes Quartus II Classic assignments, the equivalent FPGA-centric SDC constraints, and recommended system-centric SDC constraints.

Table 7–14. Quartus II Classic and Quartus II TimeQuest Equivalent Constraints

Quartus II Classic	FPGA-Centric SDC	System-Centric SDC	More Information
t_{SU} Requirement *(1)*	set_max_delay	set_input_delay -max	page 7–40
t_H Requirement *(1)*	set_min_delay	set_input_delay -min	page 7–43
t_{CO} Requirement *(2)*	set_max_delay	set_output_delay -max	page 7–45
Minimum t_{CO} Requirement *(2)*	set_min_delay	set_output_delay -min	page 7–48

Notes to Table 7–14:

(1) Refer to "t_{PD} and Minimum t_{PD} Requirement Conversion" on page 7–71 for more information about how the conversion utility converts this type of assignment.

(2) Refer to "t_{CO} Requirement Conversion" on page 7–62 for more information about how the conversion utility converts this type of assignment.

The conversion utility can convert Quartus II Classic I/O timing assignments only to the FPGA-centric constraints without additional information about your design. Making system-centric constraints requires information about the external circuitry interfacing with the FPGA such as external clocks, clock latency, board delay, and clocking exceptions. You cannot convert Quartus II Classic timing assignments to system-centric constraints without that information. If you use the conversion utility, you can modify the SDC constraints to change the FPGA-centric constraints to system-centric constraints as appropriate.

PLL Phase Shift Conversion

The conversion utility does not account for PLL phase shifts when it converts values of the following FPGA-centric I/O timing assignments:

- **t_{SU} Requirement**
- **t_H Requirement**
- **t_{CO} Requirement**
- **Minimum t_{CO} Requirement**

If any of your paths go through PLLs with a phase shift, you must correct the converted values for those paths according to the following formula:

(13)

$$\text{<correct value>} = \text{<converted value>} - \frac{(\text{<pll output period>} \times \text{<phase shift>})}{360}$$

Example 7–18 shows the incorrect conversion result for a t_{CO} assignment and how to correct it. For the example, assume the PLL output frequency is 200 MHz (period is 5 ns), the phase shift is 90 degrees, the **t_{CO} Requirement** value is 1 ns, and it is made to `data[0]`. The QSF file contains the following assignment:

Example 7–18. Assignment

```
set_instance_assignment -name TCO_REQUIREMENT -to data[0] 1.0
```

The conversion utility generates the SDC command shown in Example 7–19.

Example 7–19. SDC Command

```
set_max_delay -from [get_registers *] -to [get_ports data[0]] 1.0
```

To correct the value, use the formula and values above, as shown in the following example:

$$1.0 - \frac{(5 \times 90)}{360} = -0.25$$

Then, change the value so the SDC command looks like Example 7–20.

Example 7–20. SDC Command with Correct Values

```
set_max_delay -from [get_registers *] -to [get_ports data[0]] -0.25
```

t_{CO} Requirement Conversion

The conversion utility uses a special process to convert **t_{CO} Requirement** and **Minimum t_{CO} Requirement** assignments. In addition to the **set_max_delay** or **set_min_delay** commands, the conversion utility adds a **set_output_delay** constraint relative to a virtual clock named N/C (Not a Clock). It also creates the virtual clock named N/C with a period of 10 ns. Adding the virtual clock allows you to report timing on the output paths. Without the virtual clock N/C, the clock used for reporting would be blank. Example 7–21 shows how the conversion utility converts a **t_{CO} Requirement** assignment of 5.0 ns to **data[0]**.

Example 7–21. Converting a t_{CO} Requirement Assignment of 5.0 ns to data[0]

```
set_max_delay -from [get_registers *] -to [get_ports data[0]]
set_output_delay -clock "N/C" 0 [get_ports data[0]]
```

Entity-Specific Assignments

Next, the conversion utility converts the entity-specific assignments listed in Table 7–15 that exist in the **Timing Analyzer Settings** report panel. This usually occurs if you have any timing assignments in your Verilog HDL or VHDL source, which can include MegaCore function files. These entity-specific assignments cannot be automatically converted unless your project is compiled and a **\db** directory exists.

Table 7–15. Entity-Specific Timing Assignments

Quartus II Classic	QSF Variable	More Information
Multicycle	MULTICYCLE	page 7–37
Source Multicycle	SRC_MULTICYCLE	
Multicycle Hold	HOLD_MULTICYCLE	
Source Multicycle Hold	SRC_HOLD_MULTICYCLE	
Setup Relationship	SETUP_RELATIONSHIP	page 7–33
Hold Relationship	HOLD_RELATIONSHIP	page 7–34
Cut Timing Path	CUT	page 7–52



You must manually convert all other entity-specific timing assignments.

Paths between Unrelated Clock Domains

Beginning in version 7.1 of the Quartus II software, the conversion utility can create exceptions that cut paths between unrelated clock domains, which matches the default behavior of the Quartus II Classic Timing Analyzer. When **Cut paths between unrelated clock domains** is on, the conversion utility creates clock groups with the `set_clock_groups` command and uses the `-exclusive` option to cut paths between the clock groups.

Unsupported Instance Assignments

Finally, the conversion utility checks for the following unsupported instance assignments listed in Table 7–16 and warns you if any exist. Refer to the indicated page for information about each assignment.



You can manually convert some of the assignments to SDC constraints.

Table 7–16. Instance Timing Assignments

Assignment Name	QSF Variable	More Information
Inverted Clock	INVERTED_CLOCK	page 7–35
Maximum Clock Arrival Skew	MAX_CLOCK_ARRIVAL_SKEW	page 7–53
Maximum Data Arrival Skew	MAX_DATA_ARRIVAL_SKEW	page 7–53
Maximum Delay	MAX_DELAY	page 7–52
Minimum Delay	MIN_DELAY	page 7–52
Virtual Clock Reference	VIRTUAL_CLOCK_REFERENCE	page 7–36

Reviewing Conversion Results

You must review the messages that are generated during the conversion process, and review the SDC file for correctness and completeness. Warning and critical warning messages identify significant differences between the Quartus II Classic Timing Analyzer and Quartus II TimeQuest Timing Analyzer behaviors. In some cases, warning messages indicate that the conversion utility ignored assignments because it could not determine the intended functionality of your design. You must add to or modify the SDC constraints as necessary based on your knowledge of the design.

The conversion utility creates an SDC file with the same name as your current revision, *<revision>***.sdc**, and it overwrites any existing *<revision>***.sdc** file. If you use the conversion utility to create an SDC file, you should make additions or corrections in a separate SDC file, or a copy of the SDC file created by the conversion utility. That way, you can re-run the conversion utility later without overwriting your additions and changes. If you have constraints in multiple SDC files, refer to"Constraint File Priority" on page 7–10 to learn how to add constraints to your project.

Warning Messages

The conversion utility may generate any of the following warning messages. Refer to the information provided for each warning message to learn what to do in that instance.

Ignored QSF Variable <*assignment*>
The conversion utility ignored the specified assignment. Determine whether an equivalent constraint is necessary and manually add one if appropriate. Refer to "Timing Assignment Conversion" on page 7–33 for information about conversions for all QSF timing assignments.

Global <*name*> = <*value*>
The conversion utility ignored the global assignment <*name*>. Manually add an equivalent constraint if appropriate. Refer to "Unsupported Global Assignments" on page 7–56 for information about conversions for these assignments.

QSF: Expected <*name*> to be set to <*expected value*> but it is set to <*actual value*>
The behavior of the Quartus II TimeQuest Timing Analyzer is closest to the Quartus II Classic Timing Analyzer when the value for the specified assignment is the expected value. Because the actual assignment value is not the expected value in your project, the Quartus II TimeQuest Timing Analyzer analysis may be different from the Quartus II Classic Timing Analyzer analysis. Refer to "Recommended Global Assignments" on page 7–56 for an explanation about the indicated QSF variable names.

QSF: Found Global Fmax Requirement. Translation will be done using derive_clocks
Your design includes a global f_{MAX} requirement, and the requirement is converted to the **derive_clocks** command. Refer to "Default Required f_{MAX} Assignment" on page 7–35 for information about how to convert to an SDC constraint.

TAN Report Database not found. HDL based assignments will not be migrated
You did not analyze your design with the Quartus II Classic Timing Analyzer before running the conversion utility. As a result, the conversion utility did not convert any timing assignments in your HDL source code to SDC constraints. If you have timing assignments in your HDL source code, you must find and convert them manually, or analyze your design with the Quartus II Classic Timing Analyzer and rerun the conversion utility.

Ignored Entity Assignment (Entity <entity>): <*variable*> = <*value*> -from <*from*> -to <*to*>
The conversion utility ignored the specified entity assignment because the utility cannot automatically convert the assignment. Table 7–15 on page 7–63 lists the entity-specific assignments the script can convert automatically.

Refer to "Timing Assignment Conversion" on page 7–33 for information about how to convert the entity assignment manually.

Ignoring OFFSET_FROM_BASE_CLOCK assignment for clock <*clock*>
In some cases, this assignment is used to work around a limitation in how the Quartus II Classic Timing Analyzer handles some forms of clock inversion. The conversion script ignores the assignment because it cannot determine whether the assignment is used as a workaround. Review your clock setting and add the assignment in your SDC file if appropriate. Refer to "Clock Offset" on page 7–14 for more information about converting clock offset.

Clock <*clock*> has no FMAX_REQUIREMENT - No clock was generated
The conversion utility did not convert the clock named <*clock*> because it has no f_{MAX} requirement. You should add a clock constraint with an appropriate period to your SDC file.

No Clock Settings defined in QSF file
If your QSF file has no clock settings, ignore this message. You must add clock constraints in your SDC file manually.

Clocks

Ensure that the conversion utility converted all clock assignments correctly. Run **report_clocks**, or double-click **Report Clocks** in the Tasks pane in the Quartus II TimeQuest Timing Analyzer GUI. Make sure that the right number of clocks is reported. If any clock constraints are missing, you must add them manually with the appropriate SDC commands (**create_clock** or **create_generated_clock**). Confirm that each option for each clock is correct.

The Quartus II TimeQuest Timing Analyzer can create more clocks, such as:

- `derive_clocks` selecting ripple clocks
- `derive_pll_clocks`, adding
  - Extra clocks for PLL switchover
  - Extra clocks for LVDS pulse-generated clocks (~load_reg)

Clock Transfers

After you confirm that all clock assignments are correct, run **report_clock_transfers**, or double-click **Report Clock Transfers** in the Tasks pane in the Quartus II TimeQuest Timing Analyzer GUI. The

command generates a summary table with the number of paths between each clock domain. If the number of cross-clock domain paths seems high, remember that all clock domains are related in the Quartus II TimeQuest Timing Analyzer. You must cut unrelated clock domains. Refer to "Related and Unrelated Clocks" on page 7–13 for information about how to cut unrelated clock domains.

Path Details

If you have unexpected differences between the Quartus II Classic and Quartus II TimeQuest Timing Analyzers on some paths, follow these steps to identify the cause of the difference.

1. List the path in the Quartus II Classic Timing Analyzer.
2. Report timing on the path in the Quartus II TimeQuest Timing Analyzer.
3. Compare slack values.
4. Compare source and destination clocks.
5. Compare the launch/latch times in the Quartus II TimeQuest Timing Analyzer to the setup/hold relationship in the Quartus II Classic Timing Analyzer. The times are absolute in the Quartus II TimeQuest Timing Analyzer and relative in the Quartus II Classic Timing Analyzer.
6. Compare clock latency values.

Unconstrained Paths

Next, run `report_ucp`, or double-click **Report Unconstrained Paths** in the Tasks pane in the Quartus II TimeQuest Timing Analyzer GUI. This command generates a series of reports that detail any unconstrained paths in your design. If your design was completely constrained in the Quartus II Classic Timing Analyzer but there are unconstrained paths in the Quartus II TimeQuest Timing Analyzer, some assignments may not have been converted properly. Also, some of the assignments could be ambiguous. The Quartus II TimeQuest Timing Analyzer analyzes more paths than the Quartus II Classic Timing Analyzer does, so any unconstrained paths might be paths you could not constrain in the Quartus II Classic Timing Analyzer.

Bus Names

If your design includes Quartus II Classic Timing Analyzer timing assignments to buses, and the bus names do not include square brackets enclosing an asterisk, such as: `address[*]`, you should review the SDC constraints to ensure the conversion is correct. Refer to "Bus Name Format" on page 7–10 for more information.

Other

Review the notes listed in "Conversion Utility" on page 7–71.

Re-Running the Conversion Utility

You can force the conversion utility to run even if it can find an SDC file according to the priority described in "Constraint File Priority" on page 7–10. Any method described in "Conversion Utility" on page 7–3 forces the conversion utility to run even if it can find an SDC file.

Notes

This section describes notes for the Quartus II TimeQuest Timing Analyzer.

Output Pin Load Assignments

The Quartus II TimeQuest Timing Analyzer takes **Output Pin Load** values into account when it analyzes your design. If you change **Output Pin Load** assignments and do not recompile before you analyze timing, you must use the `-force_dat` option when you create the timing netlist. Type the following command at the Tcl console of the Quartus II TimeQuest Timing Analyzer:

```
create_timing_netlist -force_dat ↵
```

If you change **Output Pin Load** assignments and recompile before you analyze timing, do not use the `-force_dat` option when you create the timing netlist. You can create the timing netlist with the **create_timing_netlist** command, or with the **Create Timing Netlist** task in the Tasks pane.

Also note that the SDC `set_output_load` command is not supported, so you must make all output load assignments in the Quartus II Settings File (**.qsf**).

Constraint Target Types

In version 6.0 of the Quartus II software, the Quartus II TimeQuest Timing Analyzer did not support constraints between clocks and non-clocks. Beginning with version 6.1, the Quartus II TimeQuest Timing Analyzer supports mixed exception types; you can apply an exception of any clock/node combination.

DDR Constraints with the DDR Timing Wizard

The DDR Timing Wizard (DTW) creates an SDC file that contains constraints for a DDR interface. You can use that SDC file with the Quartus II TimeQuest Timing Analyzer to analyze only the DDR interface part of a design.

You should use the SDC file created by DTW for constraining a DDR interface in the Quartus II TimeQuest Timing Analyzer. Additionally, your QSF should not contain timing assignments for the DDR interface if you plan to use the conversion utility to create an SDC file. You should run the conversion utility before you use DTW, and you should choose not to apply assignments to the QSF.

However, if you used DTW and chose to apply assignments to a QSF, before you used the conversion utility, you should remove the DTW-generated QSF timing assignments and re-run the conversion utility. The conversion utility creates some incompatible SDC constraints from the DTW QSF assignments.

HardCopy Stratix Device Handoff

If you target the HardCopy device family, you should not use the Quartus II TimeQuest Timing Analyzer. The Quartus II TimeQuest Timing Analyzer is not supported for the HardCopy Stratix design process. The Quartus II TimeQuest Timing Analyzer supports HardCopy II series devices.

Unsupported SDC Features

Some SDC commands and features are not supported by the current version of the Quartus II TimeQuest Timing Analyzer. The following commands are included:

- The **get_designs** command, because the Quartus II software supports a single design, so this command is not necessary
- True latch analysis with time-borrowing feature; it can, however, convert latches to negative-edge-triggered registers
- The case analysis feature

- Loads, clock transitions, input transitions, and other features

Constraint Passing

The Quartus II software can read constraints generated by other EDA software, and write constraints to be read by other EDA software.

Other synthesis software can generate constraints that target the QSF file. If you change timing constraints in synthesis software after creating an SDC file for the Quartus II TimeQuest Timing Analyzer, you must update the SDC constraints. You can use the conversion utility, or update the SDC file manually.

Optimization

Gate-level re-timing is not supported if you turn on the Quartus II TimeQuest Timing Analyzer as your default timing analyzer.

If you use physical synthesis with the Quartus II TimeQuest Timing Analyzer, the design may have lower performance.

Clock Network Delay Reporting

In the Quartus II software version 6.0, the Quartus II TimeQuest Timing Analyzer reports delay on the clock network as a single number, rather than node-to-node segments, as the Quartus II Classic Timing Analyzer does. Beginning with version 6.0 SP1, the TimeQuest Timing Analyzer reports delay on the clock network by node-to-node segments.

PowerPlay Power Analysis

You must perform the following steps to generate an **Early Power Estimator** output file when you use the Quartus II TimeQuest Timing Analyzer and your design targets one of the following device families:

- Cyclone
- Stratix
- HardCopy Stratix

To generate an **Early Power Estimator** output file for designs targeting those families, you must perform the following steps.

1. Turn off the Quartus II TimeQuest Timing Analyzer. Refer to "Set the Default Timing Analyzer" on page 7–4 to learn how to turn off the Quartus II TimeQuest Timing Analyzer.

2. Manually convert your Quartus II TimeQuest Timing Analyzer timing constraints in the SDC file to Quartus II Classic Timing Analyzer timing assignments. You can use the Assignment Editor to enter your Quartus II Classic Timing Analyzer timing assignments in your QSF file.
3. Perform Quartus II Classic timing analysis.
4. Generate an **Early Power Estimator** output file.
5. Turn on the Quartus II TimeQuest Timing Analyzer.

Project Management

If you use the **project_open** Tcl command in the Quartus II TimeQuest Timing Analyzer to open a project compiled with an earlier version of the Quartus II software, the Quartus II TimeQuest Timing Analyzer overwrites the compilation results (**\db** folder) without warning. Opening a project any other way results in a warning, and you can choose not to open the project.

Conversion Utility

This section describes the notes for the QSF assignment to SDC constraint conversion utility.

t_{PD} and Minimum t_{PD} Requirement Conversion

The conversion utility treats the targets of single-point t_{PD} and minimum t_{PD} assignments as device outputs. It does not correctly convert targets of single-point t_{PD} and minimum t_{PD} assignments that are device inputs. The following QSF assignment applies to an a device input named `d_in`:

```
set_intance_assignment -name TPD_REQUIREMENT -to d_in "3 ns"
```

The conversion utility creates the following SDC command, regardless of whether `d_in` is a device input or device output:

```
set_max_delay "3 ns" -from [get_ports *] -to [get_ports d_in]
```

You must update any incorrect SDC constraints manually.

Referenced Documents

This chapter references the following documents:

- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *SDC and TimeQuest Tcl API Reference Manual*

Document Revision History

Table 7–17 shows the revision history for this chapter.

Table 7–17. Document Revision History (Part 1 of 2)

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No changes were made to the content.	—

May 2007 v.7.1.0	Updated chapter for Quartus II software version 7.1, including:
	• Minor changes to the "Timing Assignment Conversion" section, including:
	  • Updated data on Table 7–6 on page 7–43
	  • Updated data on Table 7–7 on page 7–45
	  • Updated data on Table 7–8 on page 7–47
	  • Updated data on Table 7–9 on page 7–49
	  • Updated data on Table 7–12 on page 7–59
	  • Added multicycle_hold information on pages 7–60 and 7–60
	• Added new section "Paths between Unrelated Clock Domains" on page 7–63
	• Added new section "Ignored Entity Assignment (Entity <entity>): <variable> = <value> -from <from> -to <to>" on page 7–65.
		Changes made to this chapter reflect the software changes made in version 7.1.

March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Minor changes made to reflect the Quartus II software version 6.1.0, including: • Changed wording on pages 7–4 and 7–5, regarding setting the default timing analyzer. • Changed Figure 7–3 on page 7–10 to reflect that the TimeQuest Timing Analyzer does not create nor convert any constraints. • Changed explanation of Figure 7–3 to reflect that TimeQuest does not create nor convert any constraints. • Changed wording in "Constraint Target Types" to reflect that the TimeQuest Timing Analyzer now supports mixed exception types.	Changes made to this chapter reflect the software changes made in version 6.1, GUI changes that were made to select the default timing analyzer, and support for mixed exception types.

Table 7–17. Document Revision History (Part 2 of 2)

Date and Version	Changes Made	Summary of Changes
July 2006 v6.0.1	Updated for the Quartus II software version 6.0.1: ● Added section on Clock Objects on page 7-24. ● Added examples of Netlist Names on page 7-29. ● Replaced figure 7-23 and example 7-9 on page 7-36. ● Added note 4 to table 7-6 on page 7-43. ● Added "Display Entity Name" to table 7-11 on page 7-57. ● Added information to Clock Conversion on pages 7-58 and 7-59. ● Added note 3 to table 7-12 on page 7-60. ● Added information to Clocks section on page 7-67. ● Added Path Details and Unconstrained Paths sections on page 7-68. ● Added information to Clock Network Delay Reporting on page 7-72. ● Added hand paragraph in Conversion Utility section on page 7-73. ● Changed "constraint" to "exception" in many places.	—
May 2006 v6.0.0	Initial release.	—

8. Quartus II Classic Timing Analyzer

QII53004-7.2.0

Introduction

Static timing analysis is a method for analyzing, debugging, and validating the timing performance of a design. The classic timing analyzer analyzes the delay of every design path and analyzes all timing requirements to ensure correct circuit operation. Static timing analysis, used in conjunction with functional simulation, allows you to verify overall design operation.

For information about switching to the Quartus II TimeQuest Timing Analyzer, refer to the *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

As part of the compilation flow, the Quartus® II software automatically performs a static timing analysis so that you do not need to launch a separate timing analysis tool. The Quartus II Classic Timing Analyzer checks every path in the design against your timing constraints for timing violations and reports results in the Timing Analysis reports, giving you immediate access to the data.

This chapter assumes you have some Tcl expertise; Tcl commands are used throughout this chapter to describe alternative methods for making timing analysis assignments. Refer to "Timing Analysis Using the Quartus II GUI" on page 8–43 for GUI-equivalent timing constraints.

This chapter details the following aspects of timing analysis:

- "Timing Analysis Tool Setup" on page 8–2
- "Static Timing Analysis Overview" on page 8–2
- "Clock Settings" on page 8–8
- "Clock Types" on page 8–9
- "Clock Uncertainty" on page 8–11
- "Clock Latency" on page 8–12
- "Timing Exceptions" on page 8–15
- "I/O Analysis" on page 8–26
- "Asynchronous Paths" on page 8–30
- "Skew Management" on page 8–34
- "Generating Timing Analysis Reports with report_timing" on page 8–36
- "Other Timing Analyzer Features" on page 8–38
- "Timing Analysis Using the Quartus II GUI" on page 8–43
- "Scripting Support" on page 8–52
- "MAX+PLUS II Timing Analysis Methodology" on page 8–58

Timing Analysis Tool Setup

The Quartus II software version 6.0 and above supports two static timing analysis tools namely, the classic timing analyzer and the Quartus II TimeQuest Timing Analyzer. Use the **Timing Analysis** option under the Settings menu to set the Timing Analyzer that is used in the compilation process.

Arria GX is not supported by the Quartus II Classic Timing Analyzer. To perform a static timing analysis for Arria GX, the Quartus II TimeQuest Timing Analyzer must be enabled.

The following steps set the classic timing analyzer as the default timing analysis tool in the Quartus II software.

1. On the Assignments menu, click **Settings**. The Settings dialog box appears.
2. In the Category list, click the icon next to **Timing Analysis Settings** to expand the folder.
3. Select the **Use Classic Timing Analyzer during compilation** radio button.

Refer to the *Quartus II TimeQuest Timing Analyzer* chapter of the *Quartus II Handbook* for more information about the Quartus II TimeQuest Timing Analyzer.

Static Timing Analysis Overview

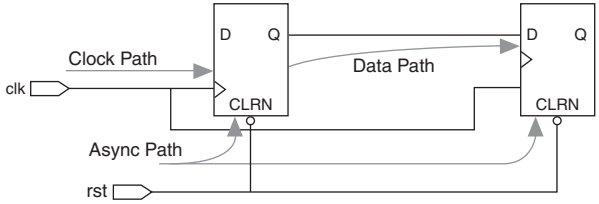
This section provides information about static timing analysis concepts used throughout this chapter and used by the Quartus II Classic Timing Analyzer. A complete understanding of the concepts presented in this section allows you to take advantage of the powerful static timing analysis features available in the Quartus II software.

Various paths exist within any given design which connect design elements together, including the path from an output of a register to the input of another register. Timing paths play a significant role during a static timing analysis. Understanding the types of timing paths is important for timing closure and optimization. Some of the commonly analyzed paths are described in this section and are shown in Figure 8–1.

- **Clock paths**—Clock paths are the paths from device pins or internally generated clocks (nodes designated as a clock via a clock setting) to the clock ports of sequential elements such as registers.
- **Data paths**—Data paths are the paths from the data output port of a sequential element to the data input port of another sequential element.

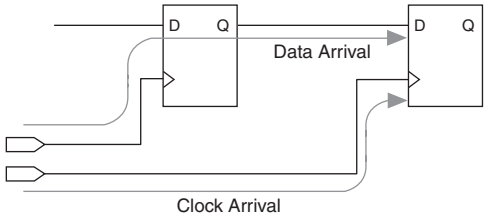
- **Asynchronous paths**—Asynchronous paths are paths from a node to the asynchronous set or clear port of a sequential element.

Figure 8–1. Path Types



Once the path types are identified, the classic timing analyzer computes data and clock arrival times for all valid register-to-register paths. Data arrival time is the delay from the source clock to the destination register. TheQuartus II Classic Timing Analyzer calculates this delay by adding the clock path delay to the source register, the micro clock-to-out (μt_{CO}) of the source register, and the data path delay from the source register to the destination register. Clock arrival time is the delay from the destination clock node to the destination register. Figure 8–2 shows a data arrival path and a clock arrival path.

Figure 8–2. Data Arrival and Clock Arrival

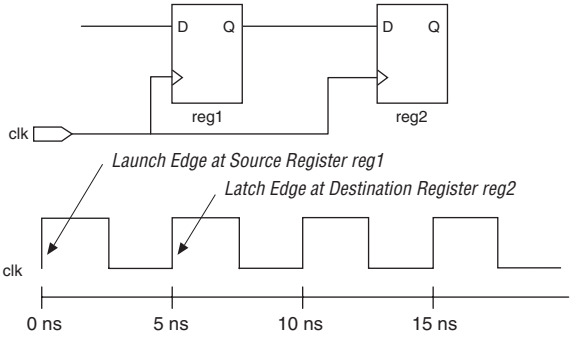


In addition to identifying various paths within a design, the Quartus II Classic Timing Analyzer analyzes clock characteristics to compute the worst-case requirement between any two registers in a single register-to-register path. You must use timing constraints to specify the characteristics of all clock signals in the design before this analysis occurs.

The active clock edge that sends data out of a sequential element, acting as a source for the data transfer, is the launch edge. The active clock edge that captures data at the data port of a sequential element, acting as a destination for the data transfer, is the latch edge.

Figure 8–3 shows a single-cycle system that uses consecutive clock edges to transmit and capture data, a register-to-register path, and the corresponding launch and latch edges timing diagram. In this example, the launch edge sends the data out of register `reg1` at 0 ns, and register `reg2` latch edge captures the data at 5 ns.

Figure 8–3. Launch Edge and Latch Edge



By analyzing specific paths relative to the launch and latch edges, the Quartus II Classic Timing Analyzer performs clock setup and clock hold checks, validating them against your timing assignments.

Clock Analysis

A comprehensive static timing analysis includes analysis of register-to-register, I/O, and asynchronous reset paths. Static Timing Analysis tools use data required times, data arrival times, and clock arrival times to verify circuit performance and detect possible timing violations. The Quartus II Classic Timing Analyzer determines the timing relationships that must be met for the design to correctly function, and checks arrival times against required times to verify timing.

Clock Setup Check

To determine if a design meets performance, the Quartus II Classic Timing Analyzer calculates clock timing, timing requirements, and timing exceptions to perform a clock setup check at each destination register based on the source and destination clocks and timing constraints, or exceptions that are applicable to those paths. A clock setup check ensures that data launched by a source register is latched correctly by the destination register. To perform a clock setup check, the Quartus II Classic Timing Analyzer determines the clock arrival time and data arrival time at the destination register by using the longest path for the

data arrival time and the shortest path for the clock arrival time. The Quartus II Classic Timing Analyzer then checks that the difference is greater than or equal to the micro setup (t_{SU}) of the destination register as shown in Equation 1.

(1) $$\text{Clock Arrival Time} - \text{Data Arrival Time} \geq \text{micro } t_{su}$$



By default, the Quartus II Classic Timing Analyzer assumes the launched and latched edges happen on consecutive active clock edges.

The results of clock setup checks are reported in terms of slack. Slack is the margin by which a timing requirement is met or not met. Positive slack indicates the margin by which a requirement is met, and negative slack indicates the margin by which a requirement is not met. The Quartus II Classic Timing Analyzer determines clock setup slack using Equations 2 through 5.

(2) $$\text{Clock Setup Slack} = \text{Data Required Time} - \text{Data Arrival Time}$$

(3) $$\text{Data Required} = \text{Clock Arrival Time} - \text{micro } t_{su} - \text{Setup Uncertainty}$$

(4) $$\text{Clock Arrival Time} = \text{Latch Edge} + \text{Shortest Clock Path to Destination Register}$$

(5) $$\text{Data Arrival Time} = \text{Launch Edge} + \text{Longest Clock Path to Source Register} + \text{micro } t_{co} + \text{Longest Data Delay}$$

The Quartus II Classic Timing Analyzer reports clock setup slack using Equations 6 through 9 (which are equivalent to Equations 2 through 5).

(6) $$\text{Clock Setup Slack} = \text{Largest Register-to-Register Requirement} - \text{Longest Register-to-Register Delay}$$

(7) $$\text{Largest Register-to-Register Requirement} = \text{Setup Relationship between Source and Destination} + \text{Largest Clock Skew} - \text{micro } t_{co} \text{ of Source Register} - \text{micro } t_{su} \text{ of Destination Register}$$

(8) $$\text{Setup Relationship between Source \& Destination Register} = \text{Latch Edge} - \text{Launch Edge} - \text{Setup Uncertainty}$$

(9) $$\text{Largest Clock Skew} = \text{Shortest Clock Path to Destination Register} - \text{Longest Clock Path to Source Register}$$

Both sets of equations can be used to determine the slack value of any path.

Clock Hold Check

To prevent hold violations, the Quartus II Classic Timing Analyzer calculates clock timing, timing requirements, and timing exceptions to perform a clock hold check at each destination register. A clock hold check ensures data launched from the source register is not captured by an active clock edge earlier than the setup latch edge, and that the destination register does not capture data launched from the next active launch edge. To perform a clock hold check, the Quartus II Classic Timing Analyzer determines the clock arrival time and data arrival time at the destination register using the shortest path for the data arrival time and the longest path for the clock arrival time. The Quartus II Classic Timing Analyzer checks that the difference is greater than or equal to the micro hold time (t_H) of the destination register, as shown in Equation 10.

(10) $$\text{Data Arrival Time} - \text{Clock Arrival Time} \geq t_H$$

The Quartus II Classic Timing Analyzer determines clock hold slack using Equations 11 through 14.

(11) $$\text{Clock Hold Slack} = \text{Data Arrival Time} - \text{Data Required Time}$$

(12) $$\text{Data Required Time} = \text{Clock Arrival Time} + \text{micro } t_H + \text{Hold Uncertainty}$$

(13) $$\text{Clock Arrival Time} = \text{Latch Edge} + \text{Longest Clock Path to Destination Register}$$

(14) $$\text{Data Arrival Time} = \text{Launch Edge} + \text{Shortest Clock Path to Source Register} + \text{micro } t_{CO} + \text{Shortest Data Delay}$$

The Quartus II Classic Timing Analyzer reports clock hold slack using Equations 15 through 18.

(15) $$\text{Clock Hold Slack} = \text{Shortest Register-to-Register Delay} - \text{Smallest Register-to-Register Requirement}$$

(16) $$\text{Smallest Register-to-Register Requirement} = \text{Hold Relationship between Source \& Destination} + \text{Smallest Clock Skew} - \text{micro } t_{CO} \text{ of Source Register} + \text{micro } t_H \text{ of Destination Register}$$

(17) $$\text{Hold Relationship between Source and Destination Register} = \text{Latch} - \text{Launch} + \text{Hold Uncertainty}$$

(18) $$\text{Smallest Clock Skew} = \text{Longest Clock Path from Clock to Destination Register} - \text{Shortest Clock Path from Clock to Source Register}$$

These equations can be used to determine the slack value of any path.

Multicycle Paths

Multicycle paths are data paths that require more than one clock cycle to latch data at the destination register. For example, a register may be required to capture data on every second or third rising clock edge. Figure 8–4 shows an example of a multicycle path between a multiplier's input registers and output register where the destination latches data on every other clock edge.

Refer to "Multicycle" on page 8–15 for more information about multicycle exceptions.

Figure 8–4. Example Diagram of a Multicycle Path

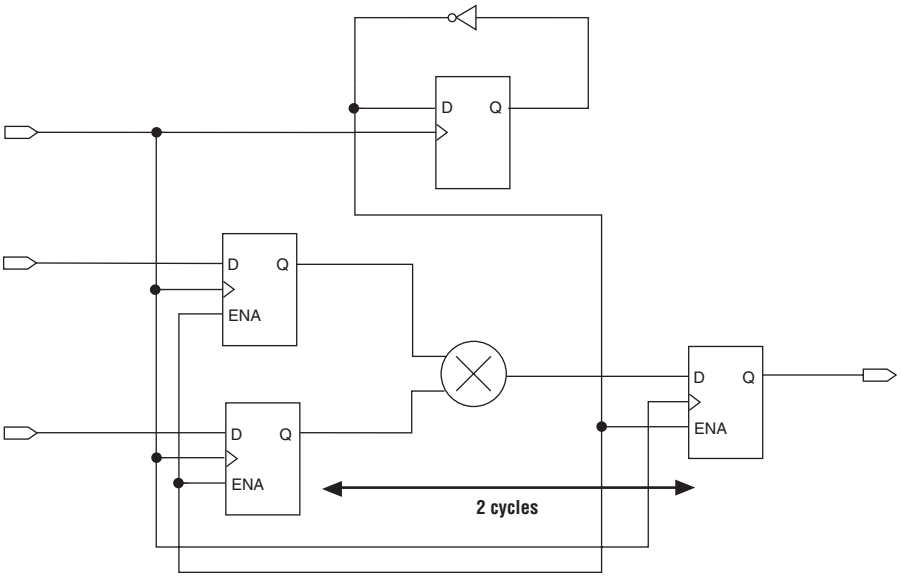


Figure 8–5 shows the default clock setup analysis launch and latch edges where multicycle assignment is equal to 1.

Figure 8–5. Default Clock Setup Analysis

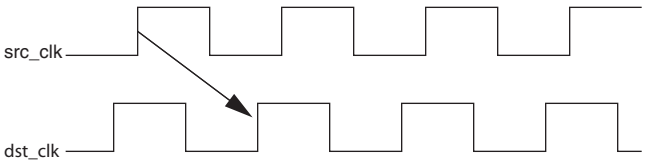
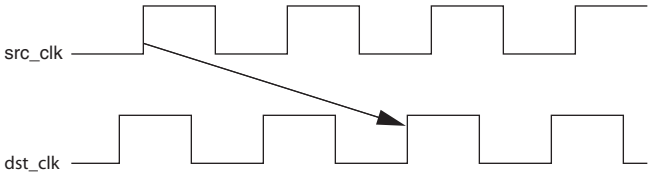


Figure 8–6 shows an analysis similar to Figure 8–5, but with a multicycle of 2.

Figure 8–6. Multicycle = 2 Clock Setup Analysis



Clock Settings

You can use individual and default clock settings to define the clocks in your design. You can base these clock settings on other clock settings already defined in your design.



To ensure the Quartus II Fitter achieves the desired performance requirements and the Quartus II Classic Timing Analyzer performs a thorough static timing analysis, you must specify all timing assignments prior to compiling the design.

Individual Clock Settings

Individual clock settings allow you to specify clock properties including performance requirements, offsets, duty cycles, and other properties for individual clock signals in your design.

You can define individual clock settings using the `create_base_clock` Tcl command. The following example defines an individual clock setting named `sys_clk` with a requirement of 100 MHz (10 ns), and assigns it to clock node `clk`.

```
create_base_clock -fmax 100MHz -target clk sys_clk
```

Default Clock Settings

You can assign a project-wide clock requirement to constrain all detected clocks in a design that do not have individual clock settings.

The `set_global_assignment -name FMAX_REQUIREMENT` Tcl command specifies a global default requirement assignment. The following example defines a 100 MHz default clock requirement:

```
set_global_assignment -name FMAX_REQUIREMENT "100.0 MHz"
```



For best placement and routing results, apply individual clock settings to all clocks in your design. All clocks adopting the default F_{MAX} are by default unrelated.

Clock Types

This section describes the types of clocks recognized by the Timing Analyzer:

- Base clocks
- Derived clocks
- Undefined clocks
- PLL clocks

Base Clocks

A base clock is independent of other clocks in a design. For example, a base clock is typically a clock signal driven directly by a device pin. A base clock is defined by individual clock settings, or automatically detected using the default clock setting.

You can use the `create_base_clock` Tcl command to define a base clock setting and assign the clock setting to a clock node. The following Tcl command creates a clock setting called `sys_clk` with a requirement of 5 ns (200 MHz) and applies the clock setting to clock node `main_clk`:

```
create_base_clock -fmax 5ns -target main_clk sys_clk
```

Derived Clocks

A derived clock is based on a previously defined base clock. For a derived clock, you can specify the phase shift, offset, multiplication and division factors, and duty cycle relative to the base clock.

You can use the `create_relative_clock` Tcl command to define and assign a derived clock setting. The following example creates a derived clock setting named `system_clockx2` that is twice as fast as the base clock `system_clock` applied to clock node `clkx2`.

```
create_relative_clock -base_clock system_clock -duty_cycle 50 \
-multiply 2 -target clkx2 system_clockx2
```

Undefined Clocks

The Quartus II Classic Timing Analyzer detects undeclared clocks in your design and displays a warning similar to the following:

```
Warning: Found pins functioning as undefined clocks and/or memory
enables
```

```
Info: Assuming node "clk_src" is an undefined clock
Info: Assuming node "clk_dst" is an undefined clock
```

The Quartus II Classic Timing Analyzer reports actual data delay for undefined clocks, but because no clock requirements exist for undefined clocks, the Quartus II Classic Timing Analyzer does not report slack for any register-to-register paths driven by an undefined clock.

PLL Clocks

Phase-locked loops (PLLs) are used for clock synthesis in Altera® devices. This device feature is configured and connected to your design using the `altpll` megafunction included with the Quartus II software. Using the MegaWizard® Plug-In Manager, you can customize the input clock frequency, multiplication factors, division factors, and other parameters of the `altpll` megafunction.

For more information about using the PLL feature in your design, refer to the *altpll Megafunction User Guide* or the handbook for the targeted device family.

For PLLs, the Quartus II Classic Timing Analyzer automatically creates derived clock settings based on the parameterization of the PLL and automatically creates a base clock setting for the input clock pin. For example, if the input clock frequency to a PLL is 100 MHz and the multiplication and division ratio is 5:2, the clock period of the PLL clock is set to 4.0 ns and is automatically calculated by the Quartus II Classic Timing Analyzer.

For the Stratix® and Cyclone® device families, you can override the PLL input clock frequency by applying a clock setting to the input clock pin of the PLL. For example, if the PLL input clock period is set to 10 ns (100 MHz) with a multiplication and division ratio of 5:2, but a clock setting of 20 ns (50 MHz) is applied to the input clock pin of the PLL, the setup relationship is 8.0 ns (125 MHz) and not 4.0 ns (250 MHz). The Quartus II Classic Timing Analyzer issues a message similar to the following:

```
Warning: ClockLock PLL
"mypll_test:inst|altpll:altpll_component|_clk1" input frequency
requirement of 200.0 MHz overrides default required fmax of 100.0
MHz -- Slack information will be reported
```

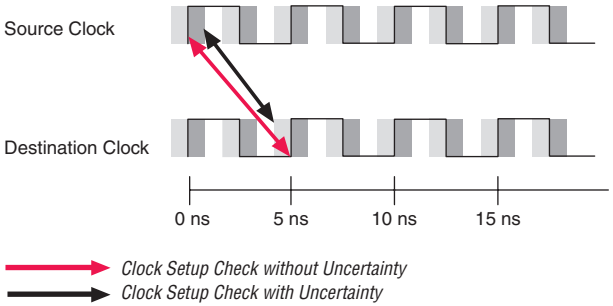
You cannot override the PLL output clock frequency with a clock setting in the Quartus II Classic Timing Analyzer.

Clock Uncertainty

You can use **Clock Setup Uncertainty** and **Clock Hold Uncertainty** assignments to model jitter, skew, or add a guard band associated with clock signals.

When a clock uncertainty assignment exists for a clock signal, the Timing Analyzer performs the most conservative setup and hold checks. For clock setup check, the setup uncertainty is subtracted from the data required time. Figure 8–7 shows an example of clock sources with a clock setup uncertainty applied.

Figure 8–7. Clock Setup Uncertainty

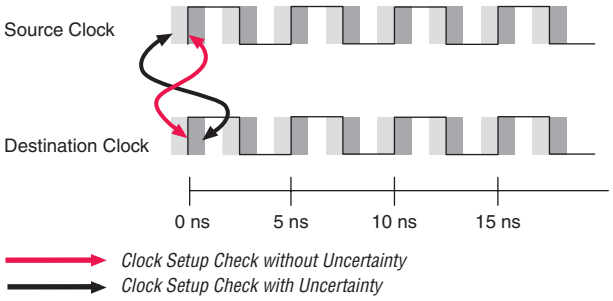


You can create clock uncertainty assignments using the Tcl command `set_clock_uncertainty`. The `set_clock_uncertainty` assignment used with the switch `-setup` specifies a clock setup uncertainty assignment. The following example creates a **Clock Setup Uncertainty** assignment with a value of 2 ns applied to clock signal `clk`:

```
set_clock_uncertainty -to clk -setup 2ns
```

For the clock hold check, the hold uncertainty is added to the data required time. Figure 8–8 shows an example of clock setup check with a clock setup uncertainty and clock hold uncertainty applied.

Figure 8–8. Clock Hold Uncertainty



You can use the `set_clock_uncertainty` Tcl command with the option `-hold` to specify a **Clock Hold Uncertainty** assignment. The following example creates a **Clock Hold Uncertainty** assignment with a value of 2 ns for clock signal `clk`.

```
set_clock_uncertainty -to clk -hold 2ns
```

You can also apply the clock uncertainty assignments between two clock sources. The following example creates a **Clock Setup Uncertainty** assignment for clock setup checks where `clk1` is the source clock and `clk2` is the destination clock:

```
set_clock_uncertainty -from clk1 -to clk2 -setup 2ns
```

Clock Latency

You can use clock latency assignments to model delays from the clock source. For example, you can use clock latency to model an external delay from an ideal clock source, such as an oscillator, to the clock pin or port of the device.

The **Early Clock Latency** assignment allows you to specify the shortest or earliest delay of the clock source. Conversely, the **Late Clock Latency** assignment allows you to specify the longest or latest delay of the clock source.

During setup analysis, the Quartus II Classic Timing Analyzer adds the **Late Clock Latency** assignment value to the source clock path delay and adds the **Early Clock Latency** assignment value to the destination clock path delay when determining clock skew for the path. During clock hold analysis, the Quartus II Classic Timing Analyzer adds the **Early Clock Latency** assignment value to the source clock path delay and adds the **Late Clock Latency** assignment value to the destination clock path delay when determining clock skew for the path.

The **Early Clock Latency** and **Late Clock Latency** assignments do not change the latch and launch edges defined by the clock setting and therefore does not change the setup or hold relationships between source and destination clocks. The clock latency assignments add only delay to the clock network and therefore only affects clock skew.

Figure 8–9 shows the clock edges used to calculate clock skew for a setup check when the **Early Clock Latency** and **Late Clock Latency** assignments are used.

Figure 8–9. Clock Setup Check Clock Skew

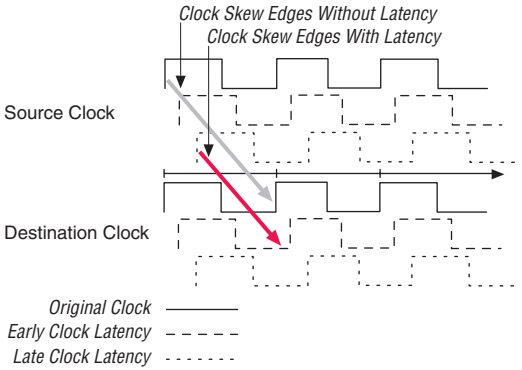
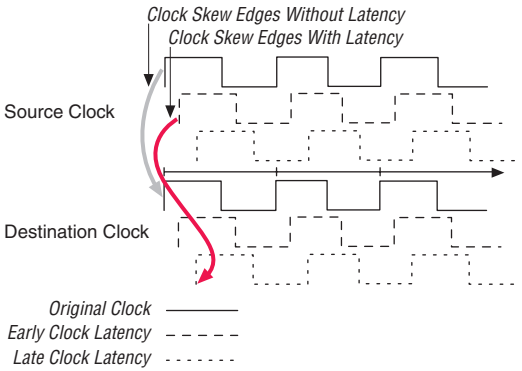


Figure 8–10 shows the clock edges used to calculate clock skew for a hold check when the **Early Clock Latency** and **Late Clock Latency** assignments are used.

Figure 8–10. Clock Hold Check Clock Skew





The Quartus II Classic Timing Analyzer ignores clock latency if the clock signal at the source and destination registers are the same.

You can use the `set_clock_latency` Tcl command with the switches `-early` or `-late` to specify an **Early Clock Latency** assignment or **Late Clock Latency** assignment, respectively. Example 8–1 specifies that the clock signal at `clk2` arrives as early as 1.8 ns and as late as 2.0 ns.

Example 8–1. Specifying Early or Late Clock Latency at clk2

```
set_clock_latency -early -to clk2 1.8ns
set_clock_latency -late -to clk2 2ns
```



The early clock latency default value is the same as the late clock latency delay, and the late clock latency default value is the same as the early clock latency delay, if only one is specified.

The **Enable Clock Latency** option must be set to **ON** for the Quartus II Classic Timing Analyzer to analyze clock latency. When this option is set to **ON**, the Quartus II Classic Timing Analyzer reports clock latency as part of the clock skew calculation for either the source or destination clock path depending upon the analysis performed. To set the **Enable Clock Latency** option to **ON**, you can use the following Tcl command:

```
set_global_assignment -name ENABLE_CLOCK_LATENCY ON
```

When the **Enable Clock Latency** option is enabled, the Quartus II Classic Timing Analyzer automatically calculates latencies for derived clocks instead of automatically calculating offsets; for example, PLL compensation delays. These clock path delays are accounted for as clock skew instead of part of the setup or hold relationship as done with offsets.



For more information about clock latency, refer to *AN 411: Understanding PLL timing for Stratix II Devices*.

Timing Exceptions

Timing exceptions allow you to modify the default behavior of the Quartus II Classic Timing Analyzer. This section describes the following timing exceptions:

- Multicycle
- Setup relationship and hold relationship
- Maximum delay and minimum delay
- False paths



Not all timing exceptions presented in this chapter are applicable to the HardCopy® II devices. If you are designing for the HardCopy II device family, refer to the *Timing Constraint for HardCopy II* chapter in the *HardCopy II Handbook*.

Multicycle

By default, the Quartus II Classic Timing Analyzer performs a single-cycle analysis for all valid register-to-register paths in the design. Multicycle assignments specify the number of clock periods before a source register launches the data or a destination register latches the data. Multicycle assignments adjust the latch or launch edges, which relaxes the required clock setup check or clock hold check between the source and destination register pairs. You can specify multicycles separately for setup and hold, and multicycles can be based on the source clock or destination clock. Apply **Multicycle** exception to time groups, clock nodes, or common clock enables.

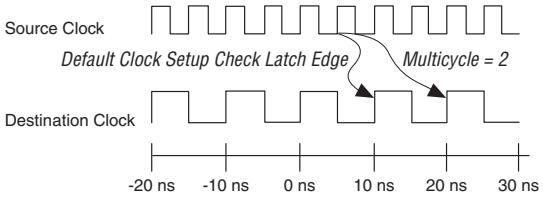
Destination Multicycle Setup Exception

A destination multicycle setup, referred to as a **Multicycle** exception, specifies the minimum number of clock cycles required before a register should latch a value. A **Multicycle** exception changes the latch edge by relaxing the required setup relationship. Figure 8–11 shows a timing diagram for a multicycle path that exists in a design with related clocks, and with the latch edge label for a clock setup check.



By default, the **Multicycle** exception value is 1.

Figure 8–11. Multicycle Setup



You can apply **Multicycle** exception between any two registers or between any two clock domains. Use the Tcl command `set_multicycle_assingment`, and the switch `–setup` and `–end`. For example, to apply a **Multicycle** exception of 2 between all registers clocked by source clock `clk_src`, and all registers clocked by destination clock `clk_dst`, enter the following Tcl command:

```
set_multicycle_assignment -setup -end -from clk_src -to clk_dst 2
```

To apply a **Multicycle** exception of 2 between the source register `reg1` and the destination register `reg2`, enter the following Tcl command:

```
set_multicycle_assignment -setup -end -from reg1 -to reg2 2
```

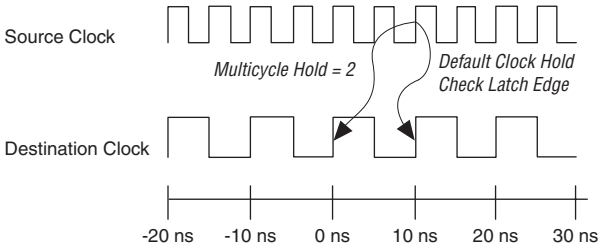
Destination Multicycle Hold Exception

A destination multicycle hold, referred to as a **Multicycle Hold** exception, modifies the latch edge used for a clock hold check for the register-to-register path based on the destination clock. A **Multicycle Hold** exception changes the latch edge by relaxing the required hold relationship. Figure 8–12 shows a timing diagram labeling the latching edge for a clock setup check.



If no **Multicycle Hold** value is specified, the **Multicycle Hold** value defaults to the value of the multicycle exception.

Figure 8–12. Multicycle Hold



You can create **Multicycle Hold** exceptions with the Tcl command `set_multicycle_assingment` and the switch `-hold` and `-end`. The following example specifies a **Multicycle Hold** exception of 3 from register `reg1` to register `reg2`:

```
set_multicycle_assignment -hold -end -from reg1 -to reg2 3
```

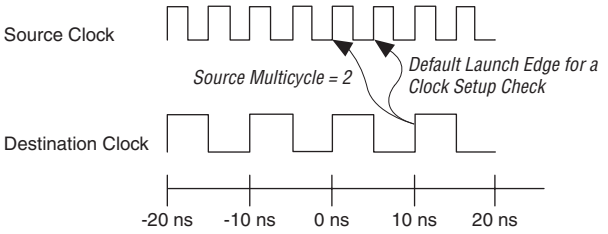
By default, the hold multicycle is set to equal that of the setup multicycle value along the same path. For example, if a setup multicycle of 2 has been applied to a register-to-register path without a separate hold multicycle, the hold multicycle value would be set to 2. The default hold multicycle value can also be changed to a value of 1. This forces all paths with a setup multicycle assignment to have a default hold multicycle of 1. To change the default hold multicycle value, in the **Settings** dialog box, click the **More Timing Settings** option.

If your design requires a hold multicycle value not equal to the setup multicycle or 1, you must explicitly apply a hold multicycle assignment to the path.

Source Multicycle Setup Exception

A source multicycle setup, referred to as **Source Multicycle Setup** exception, is used to extend the required delay by adjusting the source clock's launch edge rather than the destination clock's latch edge; for example, multicycle setup. **Source Multicycle** exceptions are useful when the source and destination registers are clocked by related clocks at different frequencies. Figure 8–13 shows an example of a **Source Multicycle** exception with the launch edge labeled for a clock setup check.

Figure 8–13. Source Multicycle



You can create **Source Multicycle Setup** exceptions with the Tcl command `set_multicycle_assignment` and the switches `-setup` and `-start`. The following example specifies a **Source Multicycle** exception of 3 from register `reg1` to register `reg2`:

```
set_multicycle_assignment -setup -start -from reg1 -to reg2 3
```

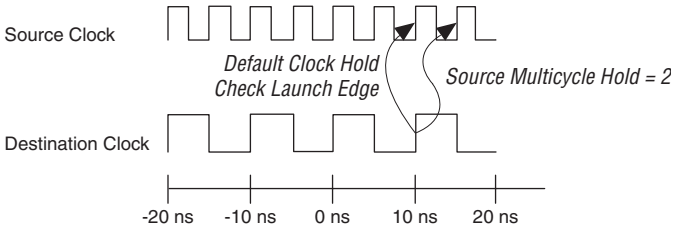
By default, the hold multicycle is set to equal that of the setup multicycle value along the same path. For example, if a setup multicycle of 2 has been applied to a register-to-register path without a separate hold multicycle, the hold multicycle value would be set to 2. The default hold multicycle value can also be changed to a value of 1. This forces all paths with a setup multicycle assignment to have a default hold multicycle of 1. To change the default hold multicycle value, in the **Settings** dialog box, click the **More Timing Settings** option.

If your design requires a hold multicycle value not equal to the setup multicycle or 1, you must explicitly apply a hold multicycle assignment to the path.

Source Multicycle Hold Exception

The **Source Multicycle Hold** exception modifies the latch edge used for a clock hold check for the register-to-register path based on the source clock. **Source Multicycle Hold** exceptions increase the required hold delay by adding source clock cycles. Figure 8–14 shows an example of a source multicycle hold with launch edge labeled for a clock hold check.

Figure 8–14. Source Multicycle Hold



You can create **Source Multicycle Hold** exceptions with the Tcl command `set_multicycle_assingment` and the switch `–setup` and `–start`. The following example specifies a **Source Multicycle Hold** exception of 3 from register `reg1` to register `reg2`:

```
set_multicycle_assignment -hold -start -from reg1 -to reg2 3
```

Default Hold Multicycle

The Quartus II Classic Timing Analyzer sets the hold multicycle value to equal the multicycle value when a multicycle exception has been entered without a corresponding hold multicycle. You can change the behavior with the `DEFAULT_HOLD_MULTICYCLE` assignment. The value of the assignment can either be `"ONE"` or `"SAME AS MULTICYCLE"`.

The assignment has the following syntax:

```
set_global_assignment -name DEFAULT_HOLD_MULTICYCLE "<value>"
```

Clock Enable Multicycle

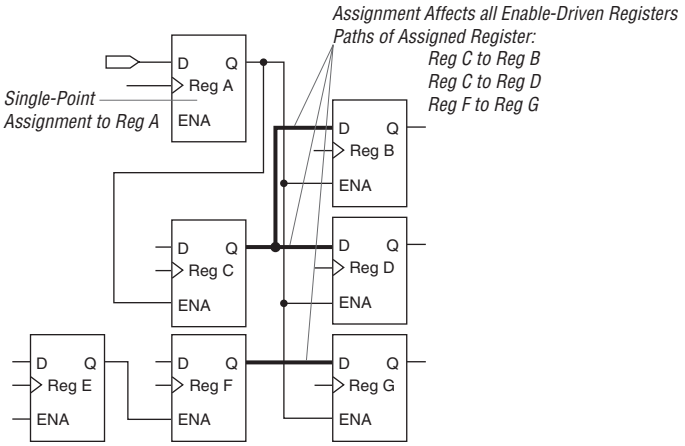
For all enable-driven registers, the setup relationship or hold relationship can be modified with the **Clock Enable Multicycle**, **Clock Enable Multicycle Hold**, **Clock Enable Source Multicycle**, or **Clock Enable Multicycle Source Hold**.

The **Clock Enable Multicycle** modifies the latching edge when a clock setup check is performed for all registers driven by the specified clock enables, and the **Clock Enable Multicycle Hold** modifies the latching edge when a clock hold check is performed for all registers driven by the specified clock enable. The **Clock Enable Source Multicycle** modifies the launching edge when a clock setup check is performed for all enabled driven registers, and the **Clock Enable Source Multicycle Hold** modifies the launching edge when a clock hold check is performed for all enabled driven registers.

Clock enable-based multicycle exceptions apply only to registers using dedicated clock enable circuitry. If the enable is synthesized into a logic cell; for example, due to signal prioritization, the multicycle does not apply.

The **Clock Enable Multicycle**, **Clock Enable Multicycle Hold**, **Clock Enable Source Multicycle**, and **Clock Enable Multicycle Source Hold** can be either a single-point or a point-to-point assignment. Figure 8–15 shows an example of a single-point assignment. In this example, register `Reg A` has the single-point assignment applied. This has the affect of modifying a register-to-register latching edge whose enable port is driven by register `Reg A`. All register-to-register paths with enables driven by the single-point assignment are affected, even those driven by different clock sources.

Figure 8–15. Single-Point Clock Enable Multicycle



Point-to-point assignments apply to all paths where the source registers' enable ports are driven by the source (`from`) node and the destination registers' enable ports are driven by the destination (`to`) node. Figure 8–16 shows an example of a point-to-point assignment made to different source and destination registers. In this example, register `Reg A` is specified as the source, and register `Reg B` is specified as the destination for the assignment. Only register-to-register paths that have their enables driven by the assigned point-to-point registers have their latching edges modified.

Figure 8–16. Different Source and Destination Point-to-Point Assignment Clock Enable Multicycle

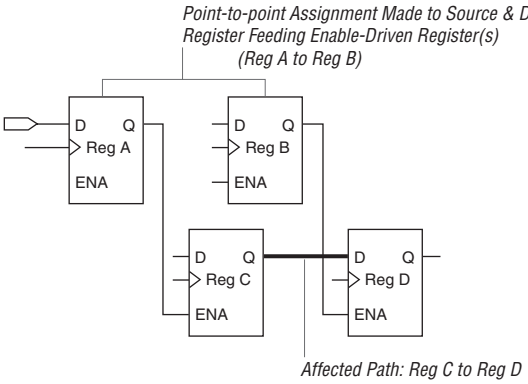
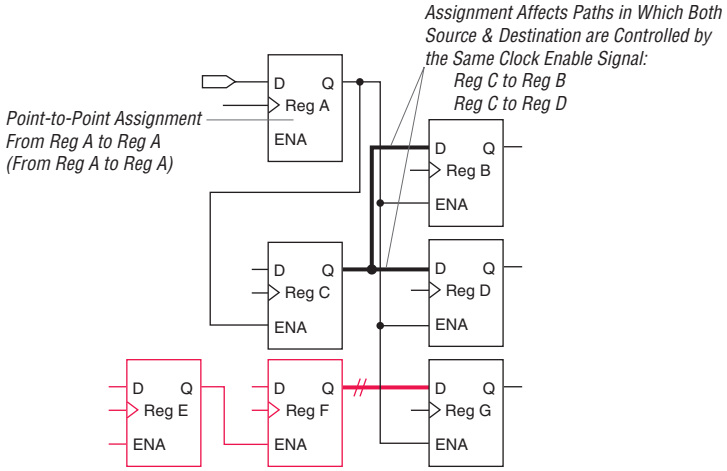


Figure 8–17 shows an example of a point-to-point assignment made to the same source and destination register. In this example, register `Reg A` has been specified as both the source and register for the assignment. Only register-to-register paths that have both the `source-enable` port and `destination-enable` port has the latching edge modified by the assigned point-to-point assignment.

Figure 8–17. Same Source and Destination Point-to-Point Assignment Clock Enable Multicycle



You can use the set_instance_assignment -name CLOCK_ENABLE_MULTICYCLE and set_instance_assignment -name CLOCK_ENABLE_MULTICYCLE_HOLD Tcl commands to specify either a **Clock Enable Multicycle** or a **Clock Enable Multicycle Hold** assignment, respectively. The following example specifies a single-point **Clock Enable Multicycle** assignment of 2 ns to reg1:

```
set_instance_assignment -name CLOCK_ENABLE_MULTICYCLE 2 -to reg1
```

The following example specifies a point-to-point **Clock Enable Multicycle Hold** assignment of 2 from register reg1 to register reg2:

```
set_instance_assignment -name CLOCK_ENABLE_MULTICYCLE_HOLD 2 \
-from reg1 -to reg2
```

You can use the set_instance_assignment -name CLOCK_ENABLE_SOURCE_MULTICYCLE and set_instance_assignment -name CLOCK_ENABLE_MULTICYCLE_SOURCE_HOLD Tcl commands to specify either a **Clock Enable Multicycle** or **Clock Enable Multicycle Hold** assignment, respectively. The following example specifies a single-point **Clock Enable Multicycle** assignment of 2 ns to reg1:

```
set_instance_assignment -name CLOCK_ENABLE_SOURCE_MULTICYCLE \
2 -to reg1
```

The following example specifies a point-to-point **Clock Enable Multicycle Hold** assignment of 2 from register reg1 to register reg2:

```
set_instance_assignment -name \
CLOCK_ENABLE_SOURCE_MULTICYCLE_HOLD 2 -from reg1 -to reg2
```

Setup Relationship and Hold Relationship

By default, the Quartus II Classic Timing Analyzer determines all setup and hold relationships based on clock settings. The **Setup Relationship** and **Hold Relationship** exceptions allow you to override any default setup or hold relationships. Example 8–2 shows the path details of a register-to-register path that has a 10 ns clock setting applied to the clock signal driving the 2 registers.

Example 8–2. Default Setup Relationship with 10 ns Clock Setting

```
Info: Slack time is 9.405 ns for clock "data_clk" between source register "reg9" and
destination register "reg10"
    Info: Fmax is restricted to 500.0 MHz due to tcl and tch limits
    Info: + Largest register to register requirement is 9.816 ns
        Info: + Setup relationship between source and destination is 10.000 ns
            Info: + Latch edge is 10.000 ns
            Info: - Launch edge is 0.000 ns
        Info: + Largest clock skew is 0.000 ns
        Info: - Micro clock to output delay of source is 0.094 ns
        Info: - Micro setup delay of destination is 0.090 ns
    Info: - Longest register to register delay is 0.411 ns
```

In Example 8–3, a 15 ns **Setup Relationship** exception is applied to the register-to-register path, overriding the default 10 ns setup relationship.

Example 8–3. Setup Relationship Assignment of 15 ns

```
Info: Slack time is 14.405 ns for clock "data_clk" between source register "reg9" and
destination register "reg10"
  Info: Fmax is restricted to 500.0 MHz due to tcl and tch limits
  Info: + Largest register to register requirement is 14.816 ns
    Info: + Setup relationship between source and destination is 15.000 ns
    Info: Setup Relationship assignment value is 15.000 ns between source "reg9" and
    destination "reg10"
    Info: + Largest clock skew is 0.000 ns
    Info: Total interconnect delay = 1.583 ns ( 51.31 % )
  Info: - Micro clock to output delay of source is 0.094 ns
  Info: - Micro setup delay of destination is 0.090 ns
Info: - Longest register to register delay is 0.411 ns
```

You can create a **Setup Relationship** exception with the Tcl command `set_instance_assignment -name SETUP_RELATIONSHIP`. The following example specifies a **Setup Relationship** exception of 5 ns from register `reg1` to register `reg2`:

```
set_instance_assignment -name SETUP_RELATIONSHIP 5ns -from reg1 \
-to reg2
```

You can use **Hold Relationship** exception to override the default hold relationship of any register-to-register paths.

You can use the `set_instance_assignment -name HOLD_RELATIONSHIP` Tcl command to specify a hold relationship assignment. The following example specifies a **Hold Relationship** exception of 1 ns from register `reg1` to register `reg2`:

```
set_instance_assignment -name HOLD_RELATIONSHIP 1ns -from reg1 \
-to reg2
```

Maximum Delay and Minimum Delay

You can use **Maximum Delay** and **Minimum Delay** assignments to specify delay requirements for pin-to-register, register-to-register, and register-to-pin paths. The **Maximum Delay** assignment overrides any setup relationship for any path. The **Minimum Delay** assignment overrides any hold relationship for any path.



The Quartus II Classic Timing Analyzer ignores the effects of clock skew when checking a design against **Maximum Delay** and **Minimum Delay** assignments.

You can use the `set_instance_assignment -name MAX_DELAY` and `set_instance_assignment -name -MIN_DELAY` Tcl commands to specify a **Maximum Delay** assignment or a **Minimum Delay** assignment, respectively. The following example specifies a maximum delay of 2 ns between source register `reg1` and destination register `reg2`:

```
set_instance_assignment -name MAX_DELAY 2ns -from reg1 -to reg2
```

The following example specifies a minimum delay of 1 ns between input pin `data_in` to destination register `dst_reg`:

```
set_instance_assignment -name MIN_DELAY 1ns -from data_in -to \
dst_reg
```

False Paths

A false path is any path that is not relevant to a circuit's operation, such as test logic. There are several global assignments to cut different classes of paths, such as unrelated clock domains and paths through bidirectional pins, but you can also cut an individual timing path to a specific false path.

The Timing Analyzer provides the following three global options that allow you to remove false paths from your design:

- Cut off feedback from I/O pins
- Cut off `read-during-write` signal paths
- Cut paths between unrelated clock domains

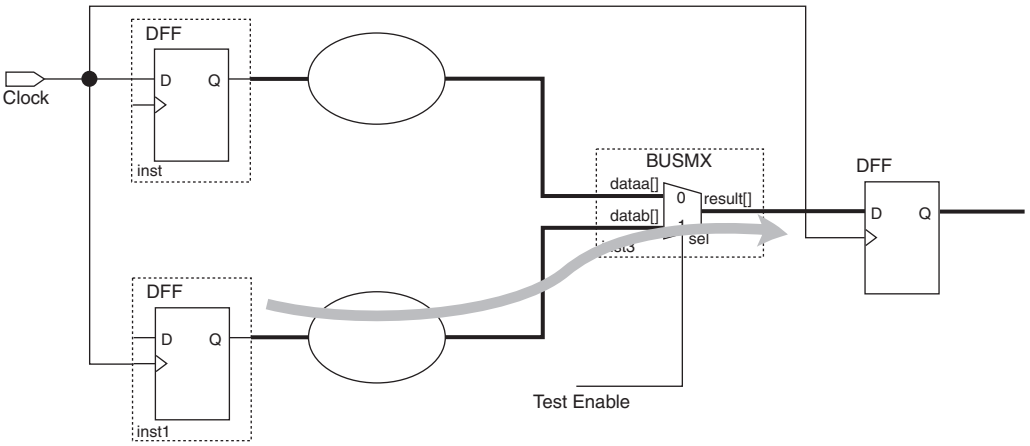
You can use the `set_global_assignment -name CUT_OFF_IO_PIN_FEEDBACK ON` Tcl command to cut the feedback path when a bidirectional I/O pin is connected directly or indirectly to both the input and output of a latch.

You can use the `set_global_assignment -name CUT_OFF_READ_DURING_WRITE_PATHS ON` Tcl command to cut the path from the `write-enable` register through memory element to a destination register.

You can use the `set_global_assignment -name CUT_OFF_PATHS_BETWEEN_CLOCK_DOMAINS ON` Tcl command to cut paths between register-to-register where the source and destination clocks are different.

You can use the `set_timing_cut_assignment` Tcl command to cut specific timing paths. In Figure 8–18, the path from `inst1` through the multiplexer to `inst2` is used only for design testing. This false path is not required under normal operation and does not need to be analyzed during static timing analysis. Figure 8–18 shows an example of a false path.

Figure 8–18. False Path Signal



To cut the timing path from source register `inst1` to destination register `inst2`, enter the following Tcl command:

```
set_timing_cut_assignment -from inst1 -to inst2
```

You can also use the `set_timing_cut_assignment` Tcl command as a single point assignment. When you use the single point assignment, all fanout of the node is cut. For example, the following Tcl command cuts all timing paths originating for node `src_reg`:

```
set_timing_cut_assignment -to src_reg
```

I/O Analysis

The I/O analysis performed by the Quartus II Classic Timing Analyzer ensures your Altera FPGA design meets all timing specifications for interfacing with external devices. This section describes assignments relevant to I/O analysis and other I/O analysis features and options available with the Quartus II Classic Timing Analyzer.

External Input Delay and Output Delay Assignments

External input and output delays represent delays from or to external devices or boards traces. You can make **Input Delay** and **Output Delay** assignments to ensure the Quartus II Classic Timing Analyzer can perform a full system analysis. By providing **Input Delays** and **Output Delays**, the Quartus II Classic Timing Analyzer is able to perform clock setup and clock hold checks for these paths. This also allows other timing assignments, such as multicycle or clock uncertainty, to be applied to input and output paths.



Do not combine individual or global t_{SU}, t_H, t_{PD}, t_{CO}, minimum t_{CO}, or minimum t_{PD} assignments with **Input Delay** or **Output Delay** assignments.

Input Delay Assignment

External input delays are specified with either **Input Maximum Delay** or **Input Minimum Delay** assignments. Make **Input Maximum Delay** assignments to specify the maximum delay of a signal from an external register to a specified input or bidirectional pin on the FPGA relative to a specified clock source. Make **Input Minimum Delay** assignments to specify the minimum delay of a signal from an external register to a specified input or bidirectional pin on the FPGA relative to a specified clock source.

When performing a clock setup check, the Quartus II Classic Timing Analyzer adds the **Input Maximum Delay** assignment value to the data arrival time (or subtracts the assignment value from the point-to-point requirement).

When performing a clock hold check, the Quartus II Classic Timing Analyzer adds the **Input Minimum Delay** assignment value to the data arrival time (or subtracts the assignment value from the point-to-point requirement).

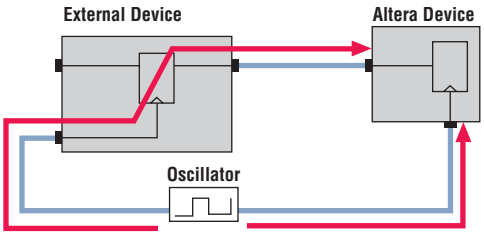
The value of the input delay assignment usually represents the sum of the t_{CO} of the external device, the actual board delay to the input pin of the Altera device, and the board clock skew.

The **Input Minimum Delay** defaults to the **Input Maximum Delay** and the **Input Maximum Delay** defaults to the Input **Minimum Delay** if only one is specified.

For example, the Input Maximum Delay and Input Minimum Delay can be used to model the delay associated with an external device driving into an Altera FPGA. Figure 8–19 shows an example of the input delay path. For Figure 8–19, the Input Maximum Delay can be calculated as shown in Equation 19.

(19) Input Maximum Delay = External Device Board Clock Path + External Device t_{CO} + External Device to Altera Device Board Delay − External Clock Path to Altera Device

Figure 8–19. Input Delay



Use the Tcl command `set_input_delay` to specify an input delay. The following example specifies an **Input Maximum Delay** assignment of 1.5 ns from clock node `clk` to input pin `data_in`:

```
set_input_delay -clk_ref clk -to "data_in" -max 1.5ns
```

The following example specifies an **Input Minimum Delay** assignment of 1 ns from clock node `clk` to input pin `data_in`:

```
set_input_delay -clk_ref clk -to "data_in" -min 1ns
```

When using **Input Delay** assignments, specify a particular clock reference. The clock reference is the clock that feeds the external register's clock port that feeds the Altera device. This allows the Quartus II Classic Timing Analyzer to perform the proper analysis for the input path.

The t_{SU}, t_H, t_{PD}, and min t_{PD} timing paths reported for input pins, where input delay internal to the Altera FPGA assignments has been applied, include only the data delay from these pins and do not account for any clock setup relationships, clock hold relationships, or slack.

Output Delay Assignment

You can specify external output delays with either **Output Maximum Delay** or **Output Minimum Delay** assignments. Make **Output Maximum Delay** assignments to specify the maximum delay of a signal from the specified FPGA output pin to an external register, relative to a specified clock source. Make **Output Minimum Delay** assignments to specify the minimum delay of a signal from the specified FPGA output pin to an external register relative to a specified clock source.

When performing a clock setup check, the Quartus II Classic Timing Analyzer subtracts the **Output Maximum Delay** assignment value from the data required time (or subtracts the assignment value from the point-to-point requirement).

When performing a clock hold check, the Quartus II Classic Timing Analyzer subtracts the **Output Minimum Delay** assignment value from the data required time (or subtracts the assignment value from the point-to-point requirement).

The value of this assignment usually represents the sum of the t_{SU} of the external device, the actual board delay from the output pin of the Altera device, and the board clock skew.

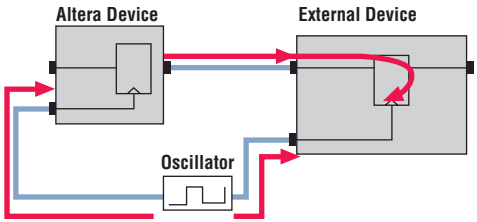


The **Output Minimum Delay** default value is the same as the **Output Maximum Delay**, and the **Output Maximum Delay** default value is the same as the **Output Minimum Delay** if only one is specified.

For example, use the **Output Maximum Delay** and **Output Minimum Delay** to model the delay associated with outputs for an Altera FPGA driving into an external device. Figure 8–20 shows an example of an output delay path. For Figure 8–20 the **Output Maximum Delay** can be calculated, as shown in Equation 20.

(20)
$$\text{Output Maximum Delay} = \text{Altera Device to External Device Board Delay} + \text{External Device } t_{SU} + \text{External Clock Path to Altera Device} - \text{External Device Board Clock Path}$$

Figure 8–20. Output Delay



The Tcl command `set_output_delay` specifies an **Output Delay** assignment. The following example specifies an **Output Maximum Delay** assignment of 2 ns from clock `clk` to output pin `data_out`:

```
set_output_delay -clk_ref clk -to data_out -max 2ns
```

The following example specifies an **Output Minimum Delay** assignment of 1 ns from clock `clk` to output pin `data_out`:

```
set_output_delay -clk_ref clk -to data_out -min 1ns
```

When using output delay assignments, specify a specific clock reference. The clock reference is the clock that feeds the external register's clock port that is fed by the Altera device. This allows the Quartus II Classic Timing Analyzer to perform the correct static timing analysis on the output path.



The t_{CO}, minimum t_{CO}, t_{PD}, and minimum t_{PD} timing paths reported for output pins, where output delay assignments have been applied include only the data delay internal to the Altera FPGA to those pins, and do not account for any clock setup relationships, clock hold relationships, or slack.

Virtual Clocks

You can use virtual clocks to model clock signals outside of the Altera FPGA, that is, clocks that do not directly drive anything within the Altera FPGA. For example, you can use a virtual clock to model a clock signal feeding an external output register that feeds the Altera FPGA.

Using the `-virtual` option of the `create_base_clock` Tcl command specifies a virtual clock assignment.



Before a you can use virtual clock for either an input or output delay assignment, the virtual clock must have the **Virtual Clock Reference** assignment enabled for the virtual clock setting.

The code in Example 8–4 creates a virtual clock named virt_clk, with a 200 MHz requirement, and uses the virtual clock setting as the clock reference for the input delay assignment.

Example 8–4. Creating a Virtual Clock Named virt_clk

```
#create the virtual clock setting
create_base_clock -fmax 200MHz -virtual virt_clk

#enable the virtual clock reference for the virtual clock setting
set_instance_assignment -name VIRTUAL_CLOCK_REFERENCE On -to virt_clk

#use the virtual clock setting as the clock reference for the input delay assignment
set_input_delay -clk_ref virt_clk -to data_in -max 2ns
```

Asynchronous Paths

The Quartus II Classic Timing Analyzer can analyze asynchronous signals that connect to the clear, preset, or load ports of a register. This section explains how the Quartus II Classic Timing Analyzer analyzes asynchronous paths.

Recovery and Removal

Recovery time is the minimum length of time an asynchronous control signal; for example, clear and preset, must be stable before the active clock edge. Removal time is the minimum length of time an asynchronous control signal must be stable after the active clock edge. The **Enable Recovery/Removal** analysis option reports the results of recovery and removal checks for paths that end at an asynchronous clear, preset, or load signal of a register.

Enable the recovery and removal analysis with the following Tcl command:

```
set_global_assignment -name ENABLE_RECOVERY_REMOVAL_ANALYSIS ON
```

With this option enabled, the Quartus II Classic Timing Analyzer reports the result of the recovery analysis and removal analysis.



By default, the recovery and removal analysis is disabled. You should enable his option for all designs that contain asynchronous controls signals.

Recovery Report

When you set ENABLE_RECOVERY_REMOVAL_ANALYSIS to **ON**, the Quartus II Classic Timing Analyzer determines the recovery time as the minimum amount of time required between an asynchronous control signal becoming inactive and the next active clock edge, compares this to your design, and reports the results as slack. The Recovery report alerts you to conditions where an active clock edge occurs too soon after the asynchronous input becomes inactive, rendering the register's data uncertain.

The recovery slack time calculation is similar to the calculation for clock setup slack, which is based on data arrival time and data required time except for asynchronous control signals. If the asynchronous control is registered, the Quartus II Classic Timing Analyzer calculates the recovery slack time using Equations 21 through 23.

(21) Recovery Slack Time = Data Required Time – Data Arrival Time

(22) Data Arrival Time = Launch Edge + Longest Clock Path to Source Register + micro t_{CO} of Source Register + Longest Register-to-Register Delay

(23) Data Required Time = Latch Edge + Longest Clock Path to Source Register + micro t_{SU} of Destination Register

Example 8–5 shows recovery time as reported by the Timing Analyzer.

Example 8–5. Recovery Time Reporting for a Registered Asynchronous Reset Signal

```
Info: Slack time is 8.947 ns for clock "a_clk" between source register "async_reg1" and destination
register "reg_1"
    Info: Requirement is of type recovery
    Info: - Data arrival time is 4.028 ns
        Info: + Launch edge is 0.000 ns
              Info: + Longest clock path from clock "a_clk" to source register is 3.067 ns
        Info: + Micro clock to output delay of source is 0.094 ns
        Info: + Longest register to register delay is 0.867 ns
    Info: + Data required time is 12.975 ns
        Info: + Latch edge is 10.000 ns
              Info: + Shortest clock path from clock "a_clk" to destination register is 3.065 ns
        Info: - Micro setup delay of destination is 0.090 ns
```

If the asynchronous control is not registered, the Quartus II Classic Timing Analyzer uses Equations 24 through Equations 26 to calculate the recovery slack time.

(24) Recovery Slack Time = Data Required Time – Data Arrival Time

(25) Data Arrival Time = Launch Edge + Maximum Input Delay + Maximum Pin-to-Register Delay

(26) Data Required Time = Latch Edge + Shortest Clock Path to Destination Register Delay – micro t_{SU} of Destination Register

Example 8–6 shows recovery time as reported by the Timing Analyzer.

Example 8–6. Recovery Time Reporting for a Non-Registered Asynchronous Reset Signal

```
Info: Slack time is 8.744 ns for clock "a_clk15" between source pin "a_arst2" and
destination register "inst5"
          Info: Requirement is of type recovery
          Info: - Data arrival time is 4.787 ns
                   Info: + Launch edge is 0.000 ns
                   Info: + Max Input delay of pin is 1.500 ns
                   Info: + Max pin to register delay is 3.287 ns
          Info: + Data required time is 13.531 ns
    Info: + Latch edge is 10.000 ns
    Info: + Shortest clock path from clock "a_clk15" to destination register
    is 3.542 ns
                  Info: - Micro setup delay of destination is 0.011 ns
```



If the asynchronous reset signal is from a device pin, an **Input Maximum Delay** assignment must be made to the asynchronous reset pin for the Quartus II Classic Timing Analyzer to perform recovery analysis on that path.

Removal Report

When you set `ENABLE_RECOVERY_REMOVAL_ANALYSIS` to **ON**, the Quartus II Classic Timing Analyzer determines the removal time as the minimum amount of time required between an active clock edge that occurs while an asynchronous input is active, and the deassertion of the asynchronous control signal. The Quartus II Classic Timing Analyzer then compares this to your design and reports the results as slack. The Removal report alerts you to a condition in which an asynchronous input signal goes inactive too soon after a clock edge, thus rendering the register's data uncertain.

The removal time slack calculation is similar to the one used to calculate clock hold slack, which is based on data arrival time and data required time except for asynchronous control signals. If the asynchronous control is registered, the Quartus II Classic Timing Analyzer uses Equations 27 through 29 to calculate the removal slack time.

(27) Removal Slack Time = Data Arrival Time – Data Required Time

(28) Data Arrival Time = Launch Edge + Shortest Clock Path from Source Register Delay + micro t_{CO} of Source Register + Shortest Register-to-Register Delay

(29) Data Required Time = Latch Edge + Longest Clock Path to Destination Register Delay + micro t_{H} of Destination Register

Example 8–7 shows removal time as reported by the Quartus II Classic Timing Analyzer.

Example 8–7. Removal Time Reporting for a Registered Asynchronous Reset Signal

```
Info: Minimum slack time is 814 ps for clock "a_clk" between source register "async_reg1"
and destination register "reg_1"
    Info: Requirement is of type removal
    Info: + Data arrival time is 4.028 ns
        Info: + Launch edge is 0.000 ns
        Info: + Shortest clock path from clock "a_clk" to source register is 3.067 ns
        Info: + Micro clock to output delay of source is 0.094 ns
        Info: + Shortest register to register delay is 0.867 ns
    Info: - Data required time is 3.214 ns
        Info: + Latch edge is 0.000 ns
       Info: + Longest clock path from clock "a_clk" to destination register is 3.065 ns
        Info: + Micro hold delay of destination is 0.149 ns
```

If the asynchronous control is not registered, the Quartus II Classic Timing Analyzer uses Equations 30 through 32 to calculate the removal slack time.

(30) Removal Slack Time = Data Arrival Time – Data Required Time

(31) Data Arrival Time = Launch Edge + Input Minimum Delay of Pin + Minimum Pin-to-Register Delay

(32) Data Required Time = Latch Edge + Longest Clock Path to Destination Register Delay + micro t_{H} of Destination Register

Example 8–8 shows removal time as reported by the Quartus II Classic Timing Analyzer.

Example 8–8. Removal Time Reporting for a Non-Registered Asynchronous Reset Signal

```
Info: Minimum slack time is 1.131 ns for clock "a_clk15" between source pin "a_arst2" and
destination register "inst5"
          Info: Requirement is of type removal
          Info: + Data arrival time is 4.787 ns
    Info: + Launch edge is 0.000 ns
    Info: + Min Input delay of pin is 1.500 ns
    Info: + Min pin to register delay is 3.287 ns
          Info: - Data required time is 3.656 ns
    Info: + Latch edge is 0.000 ns
    Info: + Longest clock path from clock "a_clk15" to destination register
    is 3.542 ns
        Info: + Micro hold delay of destination is 0.114 ns
```



If the asynchronous reset signal is from a device pin, an **Input Minimum Delay** assignment must be made to the asynchronous reset pin for the Quartus II Classic Timing Analyzer to perform a removal analysis on this path.

Skew Management

Clock skew is the difference in the arrival times of a clock signal at two different registers, which can be caused by path length differences between two clock paths, or by using gated or rippled clocks. As clock periods become shorter and shorter, the skew between data arrival times and clock arrival times becomes more significant. The Quartus II Classic Timing Analyzer provides two assignments for analyzing and constraining skew for data and clock signals.

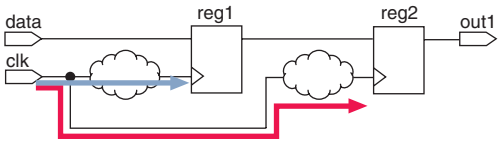
Maximum Clock Arrival Skew

Make **Maximum Clock Arrival Skew** assignments to specify the maximum allowable clock arrival skew between a clock signal and various destination registers. The Quartus II Classic Timing Analyzer compares the longest clock path to the registers' clock port and the shortest clock path to the registers' clock port to determine if your maximum clock arrival skew is achieved. Maximum clock arrival skew is calculated using Equation 33.

(33) Maximum Clock Arrival Skew = Longest Clock Path – Shortest Clock Path

For example, if the delay from clock pin `clk` to the clock port of register `reg1` is 1.0 ns, and the delay from clock pin `clk` to the clock port of register `reg2` is 3.0 ns, as shown in Figure 8–21, the Quartus II Classic Timing Analyzer provides a clock skew slack time of 2.0 ns.

Figure 8–21. Clock Arrival Paths





You should apply the **Maximum Clock Arrival Skew** assignment to a clock node and a group of registers. When you make a **Maximum Clock Arrival Skew** assignment, the Fitter attempts to satisfy the skew requirement.

You can use the `set_instance_assignment -name max_clock_arrival_skew` Tcl command to specify a **Maximum Clock Arrival Skew** assignment. The following example specifies a maximum clock arrival skew of 1 ns from clock signal `clk` to the bank of registers matching `reg*`:

```
set_instance_assignment -name max_clock_arrival_skew 1ns -from clk -to reg*
```

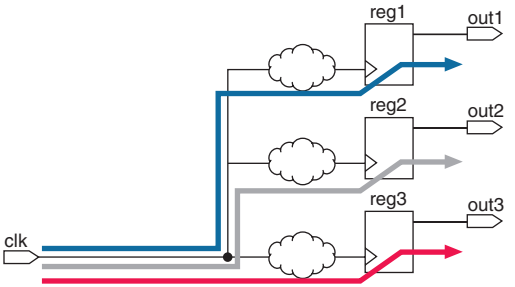
Maximum Data Arrival Skew

Make **Maximum Data Arrival Skew** assignments to specify the maximum allowable data arrival skew to various destination registers or pins. The Quartus II Classic Timing Analyzer compares the longest data arrival path to the shortest data arrival path to determine if your specified maximum data arrival skew is achieved. Maximum data arrival skew is calculated using Equation 34.

$$\text{Maximum Data Arrival Skew} = \text{Longest Data Arrival Path} - \text{Shortest Data Arrival Path} \quad (34)$$

For example, if the data arrival time to output pin `out1` is 2.0 ns, the data arrival time to output pin `out2` is 1.5 ns, and the data arrival time to output pin `out3` is 1.0 ns, as shown in Figure 8–22, the Quartus II Classic Timing Analyzer provides a maximum data arrival skew slack time of 1.0 ns.

Figure 8–22. Data Arrival Paths



When you make a **Maximum Data Arrival Skew** assignment, the Fitter attempts to satisfy the skew requirement.

You can use the `set_instance_assignment -name max_data_arrival_skew` Tcl command to specify a maximum data arrival skew value. The following example specifies a maximum data arrival skew of 1 ns from clock signal `clk` to the bank of output pins `dout`:

```
set_instance_assignment -name max_data_arrival_skew 1ns -from clk -to dout[*]
```

Generating Timing Analysis Reports with report_timing

The Quartus II Classic Timing Analyzer includes the `report_timing` Tcl command for generating text-based timing analysis reports. You can customize the output of `report_timing` using multiple switches that allow the generation of both detailed and general timing reports on any path in the design.

The `report_timing` Tcl command is available in the `quartus_tan` executable.

Prior to using the `report_timing` Tcl command, you must open a Quartus II project and create a timing netlist. For example, the following two Tcl commands accomplish this:

```
project_open my_project
create_timing_netlist
```

The report_timing Tcl command provides -from and -to switches for filtering specific source and destination nodes. For example, the following report_timing Tcl command reports all clock setup paths, with the switch –clock_setup, between registers src_reg* and dst_reg*. The –npaths 20 switch limits the report to 20 paths.

```
report_timing –clock_setup –from src_reg* -to dst_reg* -npaths 20
```

The switches -clock_filter and -src_clock_filter are also available for filtering based on specific clock sources. For example, the following report_timing Tcl command reports all clock setup paths where the destination registers are clocked by clk:

```
report_timing -clock_setup -clock_filter clk
```

The following example reports clock setup paths where the destination registers are clocked by clk, and the source registers are clocked by src_clock.

```
report_timing -clock_setup -clock_filter clk -src_clock_filter \
src_clk
```

Example 8–9 is an example script that can be sourced by the quartus_tan executable:

Example 8–9. Source for the quartus_tan Executable

```
# Open a project
project_open my_project
# Always create the netlist first
create_timing_netlist
# List clock setup paths for clock clk
# from registers abc* to registers xyz*
report_timing -clock_setup -clock_filter clk -from abc* -to xyz*
# List the top 5 pin-to-pin combinational paths
report_timing -tpd -npaths 5
# List the top 5 pin-to-pin combinational paths and
# write output to an out.tao file
report_timing -tpd -npaths 5 -file out.tao
# Compute min tpd and append results to existing out.tao
report_timing -min_tpd -npaths 5 -file out.tao -append
# Show longest path (register to register data path) between a* and b*
report_timing -longest_paths -npaths 1
delete_timing_netlist
project close
```

Other Timing Analyzer Features

The Quartus II Classic Timing Analyzer provides many features for customizing and increasing the efficiency of static timing analysis, including:

- Wildcard assignments
- Assignment groups
- Fast corner analysis
- Early timing estimation
- Timing constraint checker
- Latch analysis

Wildcard Assignments

To simplify the tasks of making assignments to many node assignments, the Quartus II software accepts the * and ? wildcard characters. Use these wildcard characters to reduce the number of individual assignments you need to make for your design.

The "*" wildcard character matches any string. For example, given an assignment made to a node specified as `reg*`, the Quartus II Classic Timing Analyzer searches and applies the assignment to all design nodes that match the prefix `reg` with none, one, or several characters following, such as `reg1`, `reg[2]`, `regbank`, and `reg12bank`.

The "?" wildcard character matches any single character. For example, given an assignment made to a node specified as `reg?`, the Quartus II Classic Timing Analyzer searches and applies the assignment to all design nodes that match the prefix `reg` and any single character following, such as `reg1`, `rega`, and `reg4`.

Assignment Groups

Assignment groups, also known as time groups, allow you to define a custom group of nodes to which you can assign timing assignments. You can also exclude specific nodes, wildcards, and time groups from a time group.

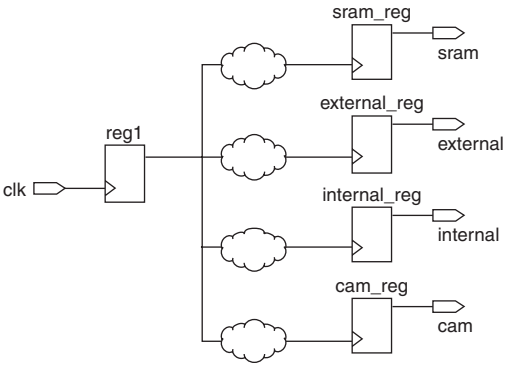
Use the `timegroup` Tcl command to create an assignment group. The following example creates an assignment group `srcgrp` and adds nodes with names that match `src1*` to the group:

```
timegroup srcgrp -add_member src1*
```

For example, Figure 8–23 has false paths between source register reg1 and destination register bank sram_reg, external_reg, internal_reg, and cam_reg that need to be cut. Without the use of assignment groups, the assignments required are:

```
set_timing_cut_assignment -from reg1 to sram_reg
set_timing_cut_assignment -from reg1 to external_reg
set_timing_cut_assignment -from reg1 to internal_reg
set_timing_cut_assignment -from reg1 to cam_reg
```

Figure 8–23. False Path



With an assignment group called dst_reg_bank, the assignments required are:

```
#create a time group called dst_reg
timegroup dst_reg_bank -add_member sram_reg
timegroup dst_reg_bank -add_member external_reg
timegroup dst_reg_bank -add_member internal_reg
timegroup dst_reg_bank -add_member cam_reg
#cut timing paths
set_timing_cut_assignment -from reg1 to dst_reg_bank
```

Once an assignment group has been defined, applicable timing assignment can be made to the time group without redefining the assignment group.

Assigning individual nodes to time groups and applying timing assignments to these time groups can improve the performance of the Quartus II Classic Timing Analyzer.

Fast Corner Analysis

Fast Corner Analysis uses timing models generated under best-case conditions (voltage, process, and temperature) for the fastest speed-grade device.



Both Fast Corner and Slow Corner static timing analysis reports are saved to the <*project name*>**.tan.rpt** file, potentially overwriting previous timing analysis reports. To preserve a copy of your reports, save the file with a new name before the next compilation or static timing analysis, or use the Combined Fast/Slow Analysis report feature.

The Quartus II software also reports minimum delay checks after a slow corner (default) analysis. These results are generated by reporting minimum delay checks using worst-case timing models.

To perform fast corner static timing analysis with the best-case timing models, you can use the `switch --fast_model=on` with the `quartus_tan` executable. The following Tcl command enables the fast timing models:

```
quartus_tan <project_name> --fast_model=on
```

Early Timing Estimation

The majority of Quartus II software compilation time is consumed by the place-and-route process used to obtain optimal design results. To accelerate the design process for large designs, the Quartus II software provides **Early Timing Estimation**. This feature provides a quick static timing analysis in a fraction of the time required for a full compilation by performing a preliminary place-and-route on the design without full optimizations, which reduces total compile time by up to five times compared to a fully fitted design.



An **Early Timing Estimate** fit is not fully optimized or legally routed. The timing delay report is only an estimate. Typically, the estimated delays are within 10% of those obtained with a full fit when the realistic setting is used.

The **Early Timing Estimate** has three settings for generating timing estimates: Realistic, Optimistic, and Pessimistic. Table 8–1 describes these settings.

Table 8–1. Early Timing Estimate Setting Options

Setting	Description
Realistic (default setting: estimates final timing using standard fitting)	Generates timing estimates that are likely to be closest to full compilation results.
Optimistic (estimates best-case final timing)	Generates timing estimates that are unlikely to be exceeded by full compilation.
Pessimistic (estimates worst-case final timing)	Generates timing estimates that are likely to be exceeded by full compilation.

To use the **Early Timing Estimate** feature, enter the following Tcl command when performing a fit:

```
quartus_fit --early_timing_estimate[=<realistic|optimistic|pessimistic>]
```

After **Early Timing Estimate** is complete, a full timing report is generated based on the early placement and routing delays. In addition, you can view the preliminary logic placement in the Timing Closure floorplan. The early timing placement allows you to perform initial placement and view the timing interaction of various placement topology.

Timing Constraint Checker

Altera recommends that you enter all timing constraints into the Quartus II software prior to performing a full compilation. This ensures that the Fitter targets the correct timing requirements and ensures that the Quartus II Classic Timing Analyzer reports the correct violations for all timing paths in the design. To ensure that all constraints have been applied to design nodes, the **Timing Constraint Check** feature reports all unconstraint paths in your design. Example 8–10 shows the timing constraint check summary generated after a full compilation.

Example 8–10. Timing Constraint Check Summary

```
+-------------------------------------------------------------------------------------------+
; Timing Constraint Check Summary                                                          ;
+---------------------------------------+---------------------------------------------------+
; Timing Constraint Check Status        ; Analyzed - Tue Feb 28 11:42:31 2006              ;
; Quartus II Version                    ; 6.1 Internal Build 143 02/20/2006 SJ Full Version ;
; Revision Name                         ; test                                              ;
; Top-level Entity Name                 ; Block1                                            ;
; Unconstrained Clocks                  ; 0                                                 ;
; Unconstrained Paths (Setup)           ; 22                                                ;
; Unconstrained Reg-to-Reg Paths (Setup) ; 0                                                ;
; Unconstrained I/O Paths (Setup)       ; 22                                                ;
; Unconstrained Paths (Hold)            ; 12                                                ;
; Unconstrained Reg-to-Reg Paths (Hold) ; 0                                                 ;
; Unconstrained I/O Paths (Hold)        ; 12                                                ;
+---------------------------------------+---------------------------------------------------+
```

To perform a timing constraint check, use the `switch --check_constraints` with the `quartus_tan` executable. The following Tcl command performs a timing constraint check on both setup and hold on the design system:

```
quartus_tan block1 --check_constraints=both
```

Latch Analysis

Latches are implemented in the Quartus II software as look-up-tables (LUTs) feeding back onto themselves. The Quartus II Classic Timing Analyzer can analyze these latches as synchronous elements rather than as combinational elements. The clock enables are analyzed as inverted clocks. The Quartus II Classic Timing Analyzer reports the results of setup and hold analysis on these latches.

You can turn on the **Analyze Latches As Synchronous Elements** option with the following Tcl command:

```
set_global_assignment -name ANALYZE_LATCHES_AS_SYNCHRONOUS_ELEMENTS ON
```

Timing Analysis Using the Quartus II GUI

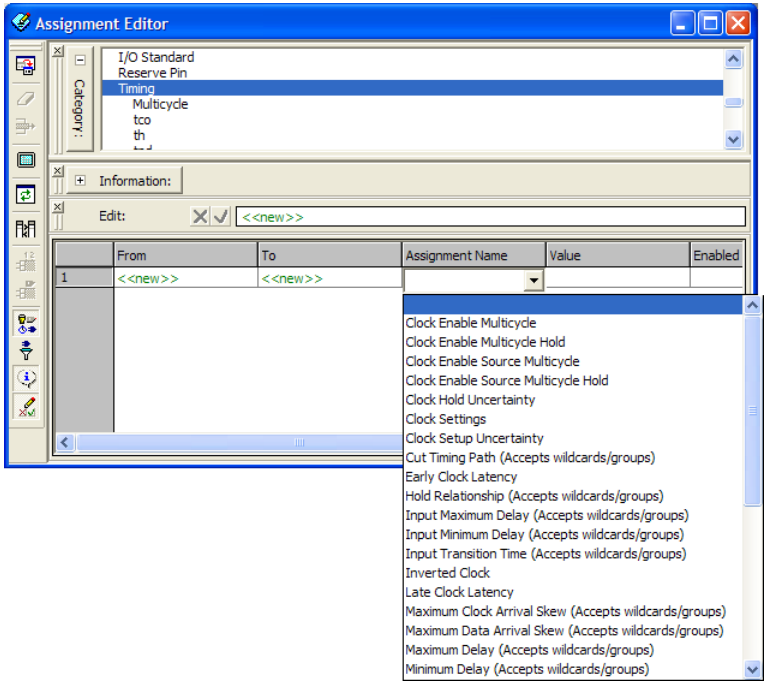
In addition to the extensive scripting support available in the Quartus II Classic Timing Analyzer, the Quartus II software provides the Assignment Editor and other user interface tools, giving you access to the Quartus II Classic Timing Analyzer features and assignments.

Assignment Editor

The Assignment Editor is a spreadsheet-style interface used for adding, modifying, and deleting timing assignments.

To make timing assignments in the Assignment Editor, choose **Timing** from the category list to cause the Assignment Name column to display only timing assignments. Double-click **<<new>>** in the **Assignment Name** field, the **Assignment Name** list displays. Figure 8–24 shows the Assignment Editor with the Assignment Name list displaying timing assignment types.

Figure 8–24. Assignment Editor



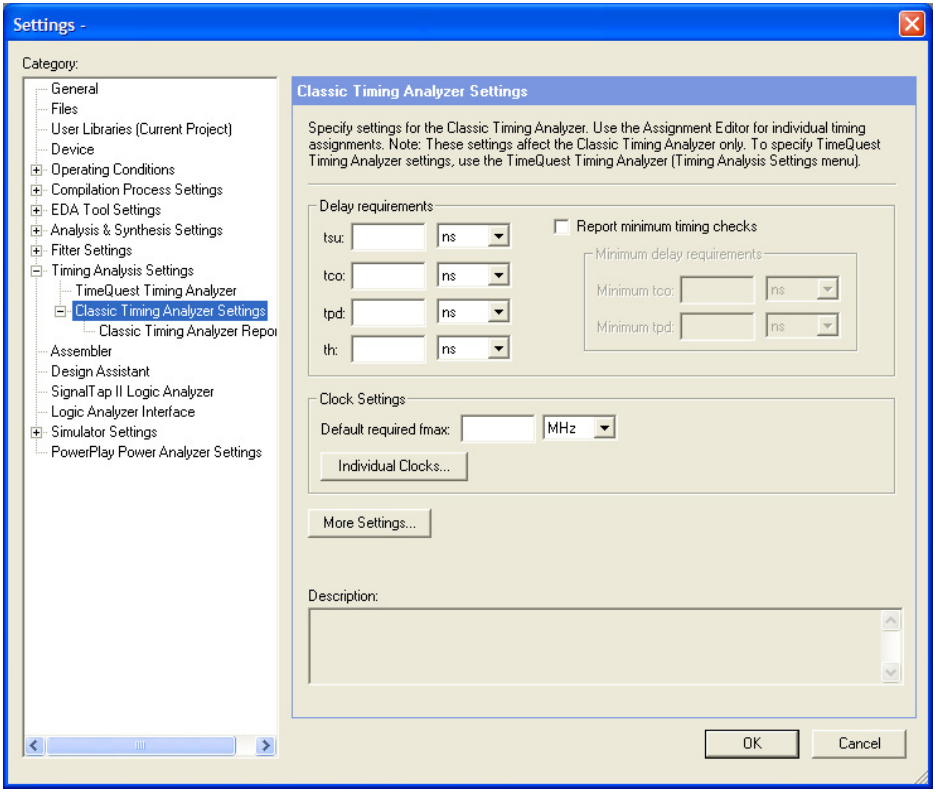
For more information about the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

Timing Settings

You can specify delay requirements and clock settings with the **Timing Analysis Settings** page of the **Settings** dialog box.

To access this page, on the Assignments menu, click **Settings**. In the Category list, click the ⊞ icon next to **Timing Analysis Settings** to expand the folder. (Be sure that the **Use Classic Timing Analyzer during compilation** radio button is turned on.) Click **Classic Timing Analyzer Settings**. The **Classic Timing Analysis Settings** page displays (Figure 8–25).

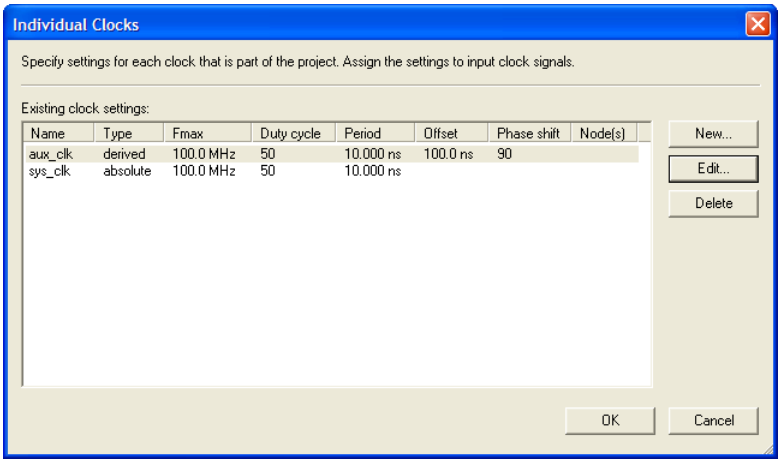
Figure 8–25. Timing Analysis Settings Dialog Box



Clock Settings Dialog Box

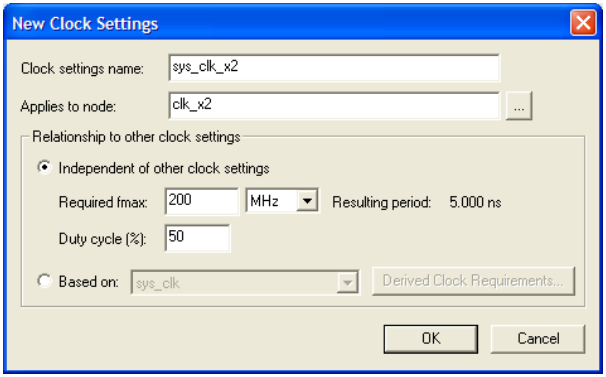
You can create or modify base clock settings or derived clock settings using the **Clock Settings** dialog box. To access this page, on the Assignments menu, click **Settings**. In the Category list, click the ⊞ icon next to **Timing Analysis Settings** to expand the folder. (Be sure that the **Use Classic Timing Analyzer during compilation** radio button is turned on.) Click on **Classic Timing Analyzer Settings**. The **Timing Analysis Settings** page displays. Under **Clock Settings**, click **Individual Clocks**. The **Individual Clock** dialog box is shown (Figure 8–26).

Figure 8–26. Individual Clocks Dialog Box



Click the **New** button in the **Individual Clocks** dialog box to access the **New Clock Settings** dialog box and create a base or derived clock setting (Figure 8–27).

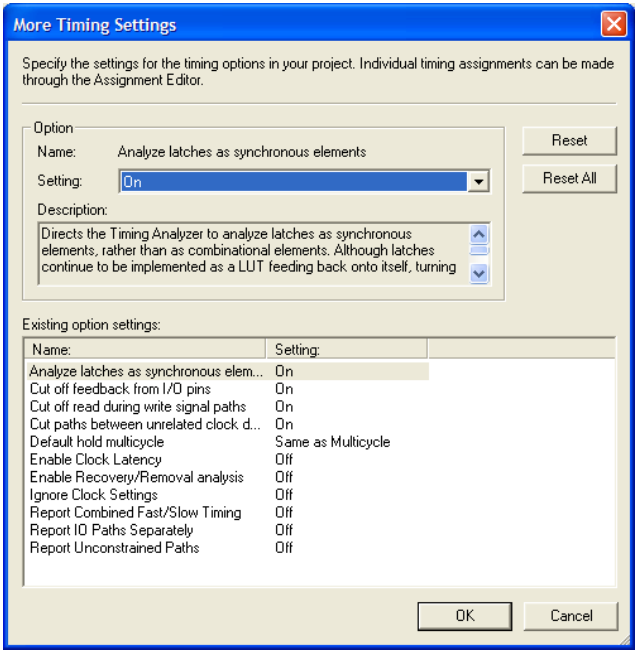
Figure 8–27. New Settings Dialog Box



More Timing Settings Dialog Box

On the **Timing Analysis Settings** page of the **Settings** dialog box, click **More Settings** to display the **More Timing Settings** dialog box (Figure 8–28). The **More Timing Settings** dialog box provides access to many global timing analysis options.

Figure 8–28. More Timing Settings Dialog Box

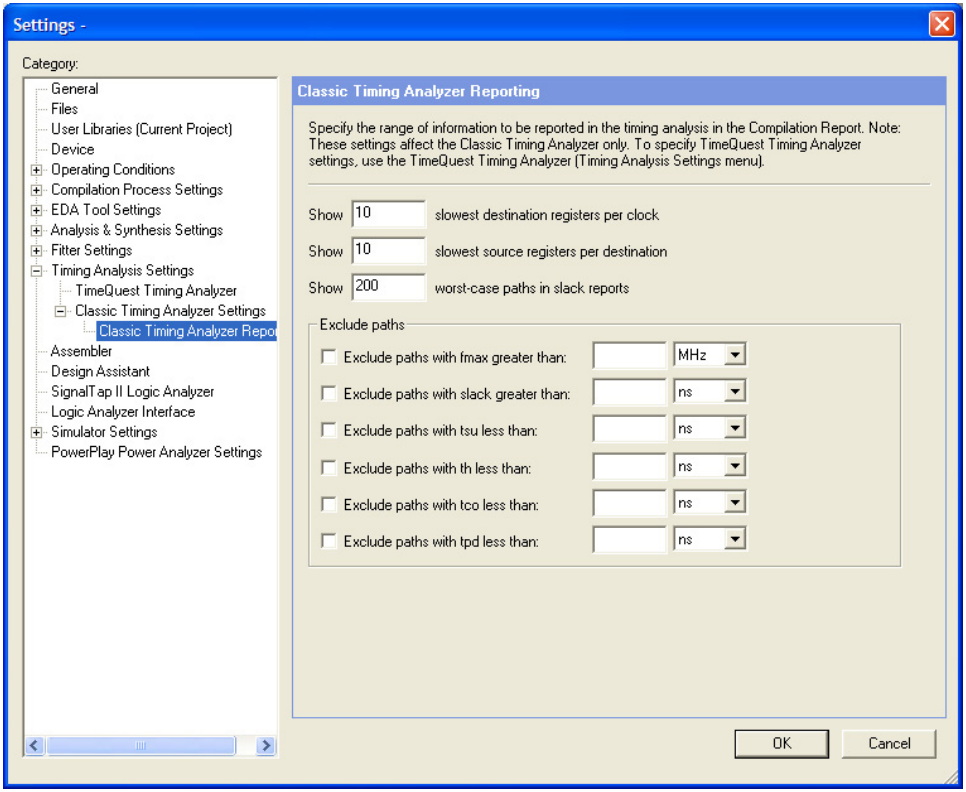


Timing Reports

The Quartus II Classic Timing Analyzer report is a section of the Compilation Report containing the static timing analysis results. The Quartus II Classic Timing Analyzer report includes clock setup and clock hold measurements for all clock sources. The report also shows t_{CO} for all output pins, t_{SU} and t_H for all input pins, and t_{PD} for any pin-to-pin combinational paths in the design. Other reports are created for different analyses and device features.

In the **Settings** dialog box, you can specify the range of information to be reported in the timing analysis of the Compilation Report. To access this page, on the Assignments menu, click **Settings**. In the Category list, click the ⊞ icon next to **Timing Analysis Settings** to expand the folder. (Be sure that the **Use Classic Timing Analyzer during compilation** radio button is turned on.) Click the ⊞ icon next to **Classic Timing Analyzer Settings** to expand the folder. Click **Classic Timing Analyzer Reporting**. The **Classic Timing Analyzer Reporting** dialog box (Figure 8–29) appears.

Figure 8–29. Classic Analyzer Reporting



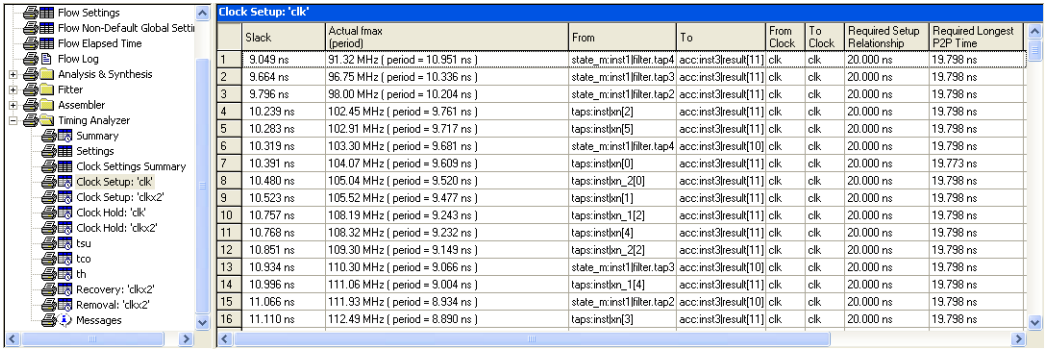
If there are no timing assignments for the design, the Quartus II Classic Timing Analyzer does not generate slack reports for any detected clock nodes. The Quartus II Classic Timing Analyzer only reports slack measurements for pins with individual or global t_{SU}, t_H, or t_{CO} assignments. A positive slack indicates the margin by which the path surpasses the clock timing requirements. A negative slack indicates the margin by which the path fails the clock timing requirements.



This Timing Analysis report is also available in text format located in the design directory with the file name *<revision name>*.**tan.rpt**.

In the Compilation Report, select an analysis type under the Timing Analyzer folder to display the analysis report; for example, Clock Setup or Clock Hold. Figure 8–30 shows an example of a Clock Setup report for clock signal `clk`.

Figure 8–30. Timing Analysis Report



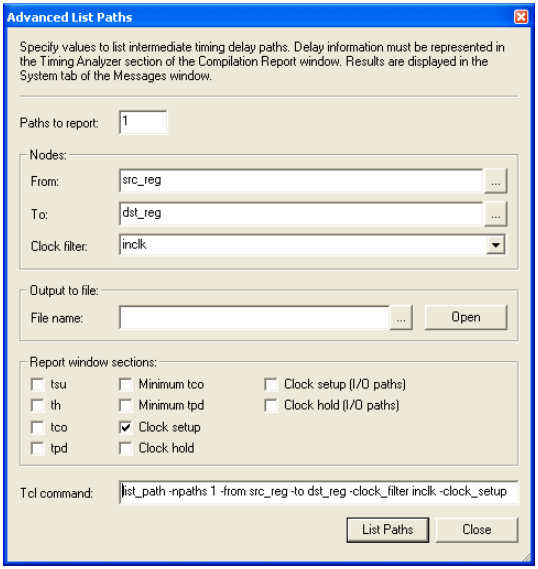
	Slack	Actual fmax (period)	From	To	From Clock	To Clock	Required Setup Relationship	Required Longest P2P Time
1	9.049 ns	91.32 MHz (period = 10.951 ns)	state_m:inst1|filter.tap4	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
2	9.664 ns	96.75 MHz (period = 10.336 ns)	state_m:inst1|filter.tap3	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
3	9.796 ns	98.00 MHz (period = 10.204 ns)	state_m:inst1|filter.tap2	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
4	10.239 ns	102.45 MHz (period = 9.761 ns)	taps:inst|xn[2]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
5	10.283 ns	102.91 MHz (period = 9.717 ns)	taps:inst|xn[5]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
6	10.319 ns	103.30 MHz (period = 9.681 ns)	state_m:inst1|filter.tap4	acc:inst3|result[10]	clk	clk	20.000 ns	19.798 ns
7	10.391 ns	104.07 MHz (period = 9.609 ns)	taps:inst|xn[0]	acc:inst3|result[11]	clk	clk	20.000 ns	19.773 ns
8	10.480 ns	105.04 MHz (period = 9.520 ns)	taps:inst|xn_2[0]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
9	10.523 ns	105.52 MHz (period = 9.477 ns)	taps:inst|xn[1]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
10	10.757 ns	108.19 MHz (period = 9.243 ns)	taps:inst|xn_1[2]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
11	10.768 ns	108.32 MHz (period = 9.232 ns)	taps:inst|xn[4]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
12	10.851 ns	109.30 MHz (period = 9.149 ns)	taps:inst|xn_2[2]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
13	10.934 ns	110.30 MHz (period = 9.066 ns)	state_m:inst1|filter.tap3	acc:inst3|result[10]	clk	clk	20.000 ns	19.798 ns
14	10.996 ns	111.06 MHz (period = 9.004 ns)	taps:inst|xn_1[4]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
15	11.066 ns	111.93 MHz (period = 8.934 ns)	state_m:inst1|filter.tap2	acc:inst3|result[10]	clk	clk	20.000 ns	19.798 ns
16	11.110 ns	112.49 MHz (period = 8.890 ns)	taps:inst|xn[3]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns

Advanced List Path

The **Advanced List Paths** dialog box provides detailed information about a specific path, such as interconnect and cell delays between any two valid register-to-register paths (Figure 8–31).

The **Advanced List Paths** dialog box allows you to select the type of paths you want listed. For example, you can obtain detailed information for Clock Setup and Clock Hold for a specific clock. In addition, the Tcl command field in the window matches the equivalent Tcl command you can use in either a custom Tcl script or in the Tcl console.

Figure 8–31. Advanced List Paths Dialog Box



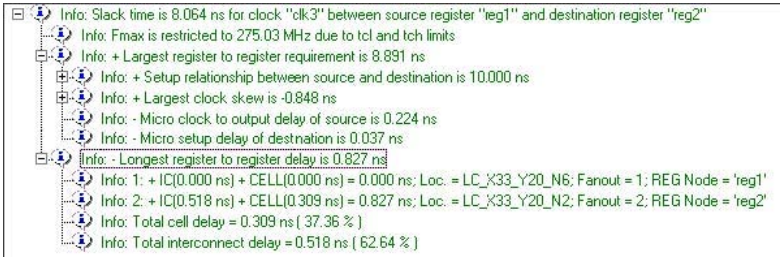
You can perform a list path command directly from the Timing Analysis report. To do this, right click a path and click **List Path** (Figure 8–32). To launch the **Advanced List Paths** dialog box, right-click a path and in the menu that appears, and select **Advanced List Paths**.

The **Advanced List Paths** dialog box displays only paths that are visible in the Timing Analysis report. To increase the amount of paths reported by the Quartus II Classic Timing Analyzer, on the Assignments menu, click **Timing Analysis Settings**. In the **Category** list, expand **Timing Analysis Settings** and select **Timing Analyzer Reporting**. In the **Timing Analyzer Reporting** page, specify the range of information to be reported by the Quartus II Classic Timing Analyzer.



Both the **Advanced List Paths** and the **List Path** commands display the path information in the **System** message window.

Figure 8–32. List Path in the Message Window



If the **Combined Fast/Slow Timing** option is enabled, the List Path Tcl command displays only path delays reported in the Slow Model section.

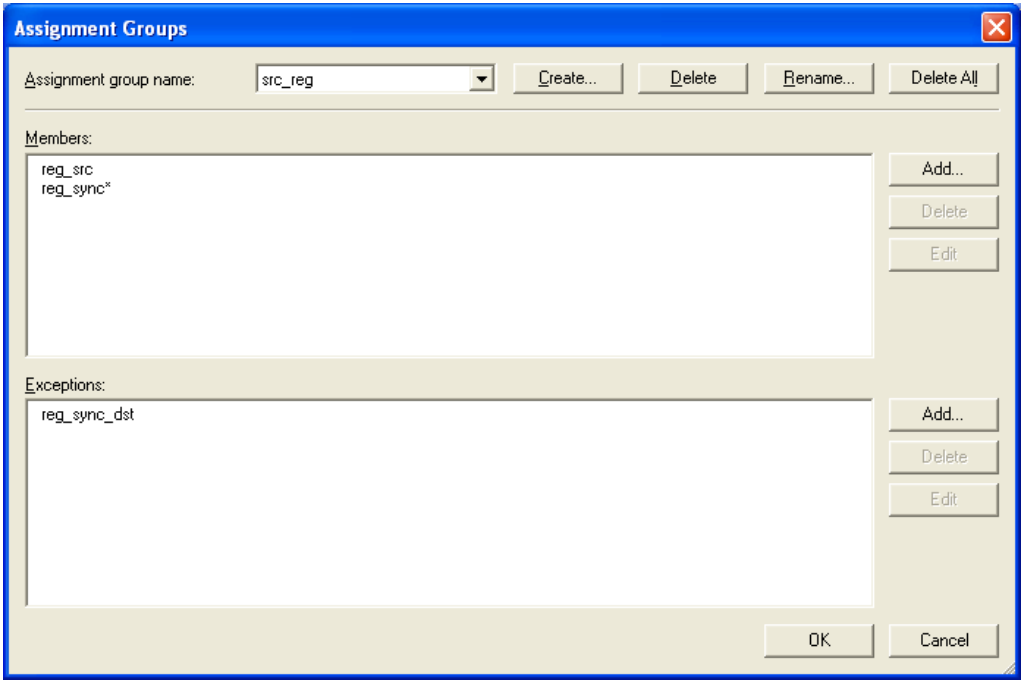
Early Timing Estimate

To start an Early Timing Estimate, on the Processing menu, point to Start and click **Start Early Timing Estimate**. To specify the **Early Timing Estimate** mode, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Processes Settings**, select **Early Timing Estimate** and click the desired timing estimate mode. For more information about the Early Timing Estimate feature, refer to "Early Timing Estimation" on page 8–40.

Assignment Groups

To define, modify, and delete assignment groups, also known as time groups, from a single dialog box, on the Assignments menu, click **Assignment (Time) Groups**. The **Assignment Groups** dialog box displays (Figure 8–33).

Figure 8–33. Assignment Groups Dialog Box



Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```

Refer to the *Scripting Reference Manual* to view this information in PDF form.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Creating Clocks

There are two Tcl commands that allow you to define clocks in a design, `create_base clock` and `create_relative_clock`.

Base Clocks

Use the `create_base_clock` Tcl command to define a base clock:

```
create_base_clock [-h | -help] [-long_help] -fmax <fmax> [-duty_cycle <integer>] \
[-virtual] [-target <name>] [-no_target] [-entity <entity>] [-disable] \
[-comment <comment>] <clock_name>
```

To define a base clock setting named `sys_clk` with a 100 MHz requirement applied to node `clk_src`, enter the following Tcl command:

```
create_base_clock –fmax 100MHz –target clk_src sys_clk
```

Derived Clocks

Use the `create_relative_clock` Tcl command to define a relative clock:

```
create_relative_clock [-h | -help] [-long_help] -base_clock <Base clock> \
[-duty_cycle <integer>] [-multiply <integer>] [-divide <integer>] [-offset <offset>] \
[-phase_shift <integer>] [-invert] [-virtual] [-target <name>] [-no_target] \
[-entity <entity>] [-disable] [-comment <comment>] <clock_name>
```

To define a relative clock named `aux_clk` based upon base clock setting `sys_clk` with a multiplication factor of 2 applied to node `rel_clk`, enter the following Tcl command:

```
create_relative_clock –base_clock sys_clk –multiply 2 –target rel_clk aux_clk
```

Clock Latency

You can use the `set_clock_latency` Tcl command to create either an early or late clock latency assignment:

```
set_clock_latency [-h | -help] [-long_help] [-early] [-late] -to <to> [<value>]
```

To apply an early clock latency of 1 ns and a late clock latency of 2 ns to clock node `clk`, enter the following Tcl commands:

```
set_clock_latency -early -to clk 2ns
```

Clock Uncertainty

You can use the set_clock_uncertainty Tcl command to create clock uncertainty assignments as shown in the following example:

```
set_clock_uncertainty [-h] [-help] [-long_help [-from <source clock name> ] -to
<destination clock name> [-setup] [-hold] [-remove] [-disable] [-comment <comment>] <value>
```

To apply a clock setup uncertainty of 50 ps between source clock node clk_src and destination clock node clk_dst, enter the following Tcl command:

```
set_clock_uncertainty -from clk_src -to clk_dst -setup 50ps
```

To apply a clock hold uncertainty of 25 ps between to clock node clk_sys, enter the following Tcl command:

```
set_clock_uncertainty -to clk_sys -setup 25ps
```

Cut Timing Paths

You can use the set_timing_cut_assignment Tcl command to create cut timing assignments:

```
set_timing_cut_assignment [-h | -help] [-long_help] [-from <from_node_list>]
[-to <to_node_list>] [-remove] [-disable] [-comment <comment>]
```

To cut the timing path from source register reg1 to destination register reg2, enter the following Tcl command:

```
set_timing_cut_assignment -from reg1 -to reg2
```

Input Delay Assignment

You can use the Tcl command set_input_delay to create input delay assignments:

```
set_input_delay [-h | -help] [-long_help] [-clk_ref <clock>] -to <input_pin> [-min] [-max]
[-clock_fall] [-remove] [-disable] [-comment <comment>] [<value>]
```

To apply an input maximum delay of 2 ns to an input pin named data_in that feeds a register clocked by clock source clk, enter the following Tcl command:

```
set_input_delay -clk_ref clk -to data_in -max 2ns
```

Maximum and Minimum Delay

The following Tcl commands create the **Maximum Delay** and **Minimum Relationship** assignments, respectively:

```
set_instance_assignment -name MAX_delay <value> -from <node> -to <node>
set_instance_assignment -name MIN_delay <value> -from <node> -to <node>
```

To apply a **Maximum Delay** of 8 ns and a minimum of 5 ns between source register `reg1` and destination register `reg2`, enter the following Tcl command:

```
set_instance_assignment -name MAX_DELAY 8ns -from reg1 -to reg2
set_instance_assignment -name MIN_DELAY 5ns -from reg1 -to reg2
```

To apply a **Maximum Delay** of 10 ns for all paths from source clock `clk_src` to destination clock `clk_dst`, enter the following Tcl command:

```
set_instance_assignment -name MAX_DELAY 10ns -from clk_src -to clk_dst
```

Maximum Clock Arrival Skew

The following Tcl command defines the **Maximum Clock Arrival Skew** assignment:

```
set_instance_assignment -name max_clock_arrival_skew <value> -from <clock> -to <node>
```

To apply a **Maximum Clock Arrival Skew** of 1 ns for clock source `clk` to a predefined timegroup called `reg_group`, enter the following Tcl command:

```
set_instance_assignment -name max_clock_arrival_skew 1ns -from clk -to reg_group
```

Maximum Data Arrival Skew

To create **Maximum Data Arrival Skew** assignments, use the Tcl command `set_instance_assignment -name max_data_arrival`:

```
set_instance_assignment -name max_data_arrival_skew <value> -from <clock> -to <node>
```

To apply a **Maximum Data Arrival Skew** of 1 ns for clock source `clk` to a predefined timegroup of pins called `pin_group`, enter the following Tcl command:

```
set_instance_assignment -name max_data_arrival_skew 1ns -from clk -to pin_group
```

Multicycle

Use the `set_multicycle_assignment` Tcl command to create **Multicycle** assignments:

```
set_multicycle_assignment [-h | -help] [-long_help] [-setup] [-hold] [-start] [-end]
[-from <from_list>] [-to <to_list>] [-remove] [-disable] [-comment <comment>]
<path_multiplier>
```

To apply a **Multicycle Setup** of 2 and a **Hold Multicycle** of 1 between source register `reg1` and destination register `reg2`, enter the following Tcl commands:

```
set_multicycle_assignment –setup -end –from reg1 –to reg2 2
set_multicycle_assignment –hold -end –from reg1 –to reg2 1
```

To apply a **Source Multicycle Setup** of 2 between source register `reg1` and destination register `reg2`, enter the following Tcl command:

```
set_multicycle_assignment –setup -start –from reg1 –to reg2 1
```

To apply a **multicycle setup** of 2 for all paths from source clock `clk_src` to destination clock `clk_dst`, enter the following Tcl command:

```
set_multicycle_assignment –setup –end –from clk_src –to clk_dst 2
```

Output Delay Assignment

Use the Tcl command `set_output_delay` to create **Output Delay** assignments:

```
set_output_delay [-h | -help] [-long_help] [-clk_ref <clock>] -to <output_pin> [-min]
[-max] [-clock_fall] [-remove] [-disable] [-comment <comment>] [<value>]
```

To apply an **Output Maximum Delay** of 3 ns to an output pin named `data_out` that is fed to a register clocked by clock source `clk`, enter the following Tcl command:

```
set_output_delay -clk_ref clk -to data_out –max 3ns
```

Report Timing

Use the `report_timing` Tcl command to generate timing reports:

```
report_timing [-h | -help] [-long_help] [-npaths <number>] [-tsu] [-th] [-tco] [-tpd] \
[-min_tco] [-min_tpd] [-clock_setup] [-clock_hold] [-clock_setup_io] [-clock_hold_io] \
[-clock_setup_core] [-clock_hold_core] [-recovery] [-removal] [-dqs_read_capture] \
[-stdout] [-file <name>] [-append] [-table <name>] [-from <names>] [-to <names>] \
[-clock_filter <names>] [-src_clock_filter <names>] [-longest_paths] [-shortest_paths] \
[-all_failures]
```

The following example generates a list of all clock setup paths for clock source `clk` from registers `src_reg*` to registers `dst_reg*`:

```
report_timing -clock_setup -clock_filter clk -from src_reg* -to dst_reg*
```

Setup and Hold Relationships

The following Tcl commands create **Setup Relationship** and **Hold Relationship** assignments, respectively:

```
set_instance_assignment -name SETUP_RELATIONSHIP <value> -from <node> -to <node>
set_instance_assignment -name HOLD_RELATIONSHIP <value> -from <node> -to <node>
```

To apply a **Setup Relationship** of 12 ns and a **Hold Relationship** of 2 ns between source register `reg1` and destination registers `reg2`, enter the following Tcl command:

```
set_instance_assignment -name SETUP_RELATIONSHIP 12ns -from reg1 -to reg2
set_instance_assignment -name HOLD_RELATIONSHIP 2ns -from reg1 -to reg2
```

To apply a setup relationship of 10 ns for all paths from source clock `clk_src` to destination clock `clk_dst`, enter the following Tcl command:

```
set_instance_assignment -name SETUP_RELATIONSHIP 10ns -from clk_src -to clk_dst
```

Assignment Group

Use the `timegroup` Tcl command to create assignment groups:

```
timegroup [-h | -help] [-long_help] [-add_member <name>] [-add_exception <name>] \
[-remove_member <name>] [-remove_exception <name>] [-get_members] [-get_exceptions] \
[-overwrite] [-remove] [-disable] [-comment <comment>] <group_name>
```

The following example creates an assignment group called `reg_bank` with members `dst_reg*`, and excludes register `dst_reg5`.

```
timegroup reg_bank -add_member dst_reg* -add_exception dst_reg5
```

Virtual Clock

Use the `create_relative_clock` with the `-virtual` switch to create **Virtual Clock** assignments:

```
create_relative_clock [-h | -help] [-long_help] -base_clock <Base clock> \
[-duty_cycle <integer>] [-multiply <integer>] [-divide <integer>] [-offset <offset>] \
[-phase_shift <integer>] [-invert] [-virtual] [-target <name>] [-no_target] \
[-entity <entity>] [-disable] [-comment <comment>] <clock_name>
```

To define a virtual clock derived from the base clock setting `clk_aux` named `brd_sys`, enter the following Tcl command:

```
create_relative_clock -base_clock clk_aux -virtual brd_sys
```

MAX+PLUS II Timing Analysis Methodology

This section describes the basic static timing analysis and assignments available in the Quartus II software that originated in the MAX+PLUS® II design software.

f_{MAX} Relationships

Maximum clock frequency is the fastest speed at which the design clock can run without violating internal setup and hold time requirements. The Quartus II software performs static timing analysis on both single- and multiple-clock designs.



Apply clock settings to all clock nodes in a design to ensure that you meet all performance requirements. Refer to "Clock Settings" on page 8–8 for more information.

Slack

Slack is the margin by which a timing requirement such as f_{MAX} is met or not met. Positive slack indicates the margin by which a requirement is met. Negative slack indicates the margin by which a requirement is not met. The Quartus II software determines slack using Equations 35 through 38.

(35) Clock Setup Slack = Longest Register-to-Register Requirement – Longest Register-to-Register Delay

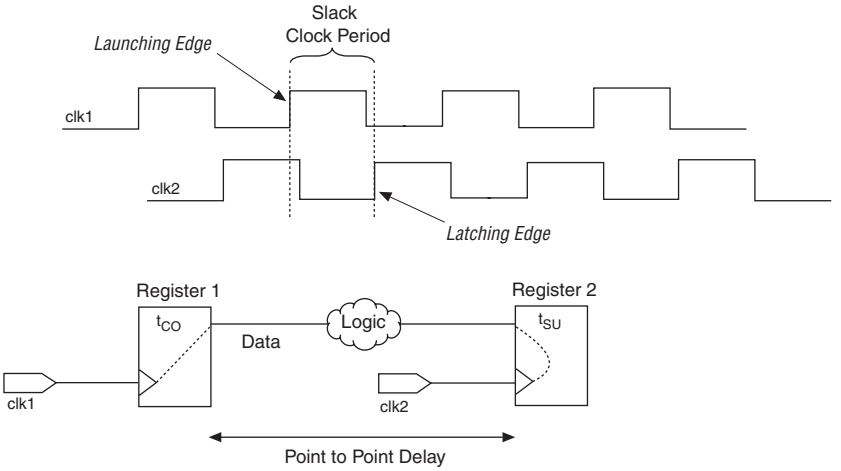
(36) Register-to-Register Requirement = Setup Relationship + Largest Clock Skew – micro t_{CO} of Source Register – micro t_{SU} of Destination Register

(37) Clock Hold Slack = Shortest Register-to-Register Delay – Smallest Register-to-Register Requirement

(38) Shortest Register-to-Register Requirement = Hold Relationship + Smallest Clock Skew – micro t_{CO} of Source Register – micro t_{H} of Destination Register

Figure 8–34 shows a slack calculation diagram.

Figure 8–34. Slack Calculation Diagram



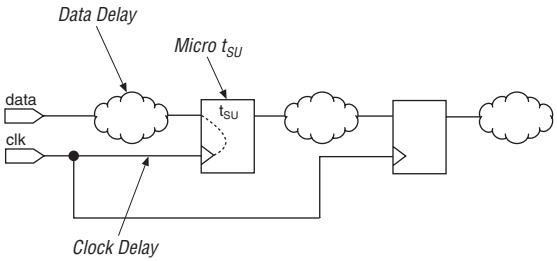
I/O Timing

This section describes the basic measurements made for I/O timing in the Quartus II software.

t_{SU} Timing

t_{SU} specifies the length of time data needs to arrive and be stable at an external input pin prior to a clock transition on an associated clock I/O pin. A t_{SU} requirement describes this relationship for an input register relative to the I/O pins of the FPGA. Figure 8–35 shows a diagram of clock setup time.

Figure 8–35. Clock Setup Time (t_{SU})



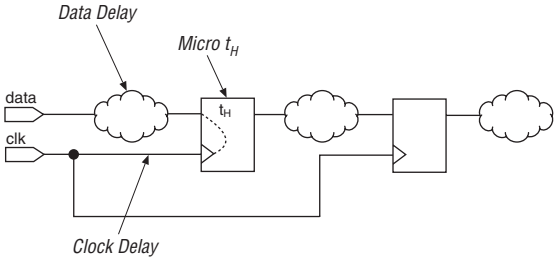
Micro t_{SU} is the internal setup time of the register. It is a characteristic of the register and is unaffected by the signals feeding the register. Equation 39 calculates the t_{SU} of data with respect to `clk` for the circuit shown in Figure 8–35.

(39) t_{SU} = Longest Data Delay – Shortest Clock Delay + micro t_{SU} of Input Register

t_H Timing

t_H specifies the length of time data needs to be held stable on an external input pin after a clock transition on an associated clock I/O pin. A t_H requirement describes this relationship for an input register relative to the I/O pins of the FPGA. Figure 8–36 shows a diagram of clock hold time.

Figure 8–36. Clock Hold Time (t_H)



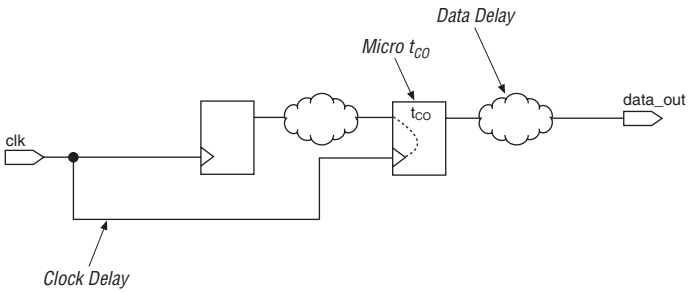
Micro t_H is the internal hold time of the register. Equation 40 calculates the t_H of data with respect to `clk` for the circuit shown in Figure 8–36.

(40) t_H = Longest Clock Delay – Shortest Data Delay + micro t_H of Input Register

t_{CO} Timing

Clock-to-output delay is the maximum time required to obtain a valid output at an output pin fed by a register, after a clock transition on the input pin that clocks the register. Micro t_{CO} is the internal clock-to-output delay of the register. Figure 8–37 shows a diagram of clock-to-output delay.

Figure 8–37. Clock-to-Output Delay (t_{CO})



Equation 41 calculates the t_{CO} for output pin `data_out` with respect to clock node `clk` for the circuit shown in Figure 8–37.

(41) t_{CO} = Longest Clock Delay + micro t_{CO} of Output Register

Minimum t_{CO} (min t_{CO})

Minimum clock-to-output delay is the minimum time required to obtain a valid output at an output pin fed by a register, after a clock transition on the input pin that clocks the register. Micro t_{CO} is the internal clock-to-output delay of registers in Altera FPGAs. Unlike the t_{CO} assignment, the min t_{CO} assignment looks at the shortest delay paths (Equation 42).

(42) $\text{min } t_{CO} = \text{Shortest Clock Delay} + \text{Shortest Data Delay} + \text{micro } t_{CO} \text{ of Output Register}$

t_{PD} Timing

Pin-to-pin delay (t_{PD}) is the time required for a signal from an input pin to propagate through combinational logic and appear at an external output pin (Equation 43).

(43) $t_{PD} = \text{Longest Pin-to-Pin Delay}$



In the Quartus II software, you can make t_{PD} assignments between an input pin and an output pin.

Minimum t_{PD} (min t_{PD})

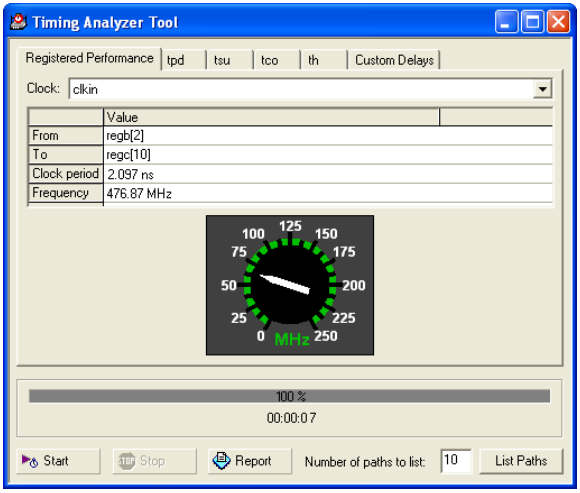
The minimum pin-to-pin delay (t_{PD}) is the time required for a signal from an input pin to propagate through combinational logic and appear at an external output pin. Unlike the t_{PD} assignment, the min t_{PD} assignment applies to the shortest pin-to-pin delay (Equation 44).

(44) $\text{min } t_{PD} = \text{Shortest Pin-to-Pin Delay}$

The Timing Analyzer Tool

To facilitate the classic static timing analysis flow and constraint, the Quartus II software provides a MAX+PLUS II-style Timing Analyzer Tool available on the Tools menu. The Timing Analyzer Tool provides a simple interface, similar to the Timing Analyzer tool in MAX+PLUS II, that reports register-to-register performance, I/O timing, and custom delay values (Figure 8–38).

Figure 8–38. Timing Analyzer Tool



Conclusion

Evolving design and aggressive process technologies require larger and higher-performance FPGA designs. Increasing design complexity demands enhanced static timing analysis tools that aid designers in verifying design timing requirements. Without advanced static timing analysis tools, you risk circuit failure in complex designs. The Quartus II Classic Timing Analyzer incorporates a set of powerful static timing analysis features critical in enabling system-on-a-programmable-chip (SOPC) designs.

Referenced Documents

This chapter references the following documents:

- *altpll Megafunction User Guide*
- *AN 411: Understanding PLL Timing for Stratix II Devices*
- *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter of the *Quartus II Handbook*
- *Scripting Reference Manual*

Document Revision History

Table 8–2 shows the revision history for this chapter.

Table 8–2. Document Revision History

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 8–63.	—
May 2007 v7.1.0	• Updated Quartus II software 7.1 revision and date • Added information about Arria GX • Added Referenced Document • No new screenshots were taken	Very minor update pertaining to Arria GX.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	• Added paragraphs about multicycle assignments on page 8–17 and page 8–18 • Updated Figure 8–24 on page 8–42 (screenshot update) • Updated Figure 8–25 on page 8–43 (screenshot update)	Minor clarification of text referring to input and output delay assignments.
May 2006 v6.0.0	Chapter title changed to *classic timing analyzer*. Updated for the Quartus II software version 6.0.0: • Updated GUI information.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
August 2005 V5.0.1	Document revision 1.0.	—
May 2005 V5.0.0	New functionality for Quartus II software 5.0	—
Jan. 2005 v2.2	Updated information pertaining to realistic, optimistic, and pessimistic settings	—
Dec. 2004 v2.1	• Chapter 5 was formerly Chapter 4. • Updates to tables and figures. • New functionality for Quartus II software 4.2.	—
June 2004 v2.0	• Updates to tables and figures. • New functionality for Quartus II software 4.1.	—
Feb. 2004 v1.0	Initial release.	—
May 2006 v6.0.0	Chapter title changed to *Classic Timing Analyzer*. Updated for the Quartus II software version 6.0.0: • Updated GUI information.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—

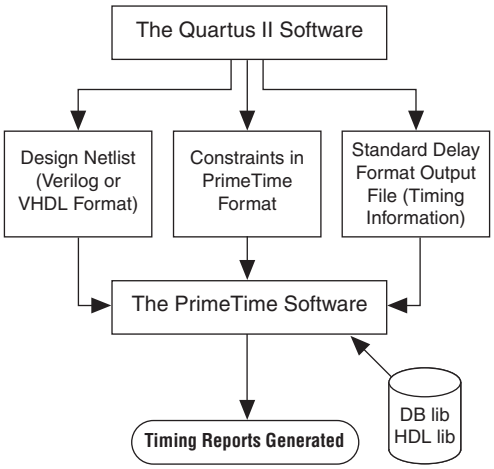
9. Synopsys PrimeTime Support

QII53005-7.2.0

Introduction

PrimeTime is an industry standard sign-off tool that performs static timing analysis on ASIC designs. The Quartus® II software makes it easy for designers to analyze their Quartus II projects using the PrimeTime software. The Quartus II software exports a netlist, design constraints (in the PrimeTime format), and libraries to the PrimeTime software environment. Figure 9–1 shows the PrimeTime flow diagram.

Figure 9–1. The PrimeTime Software Flow Diagram



This chapter contains the following sections:

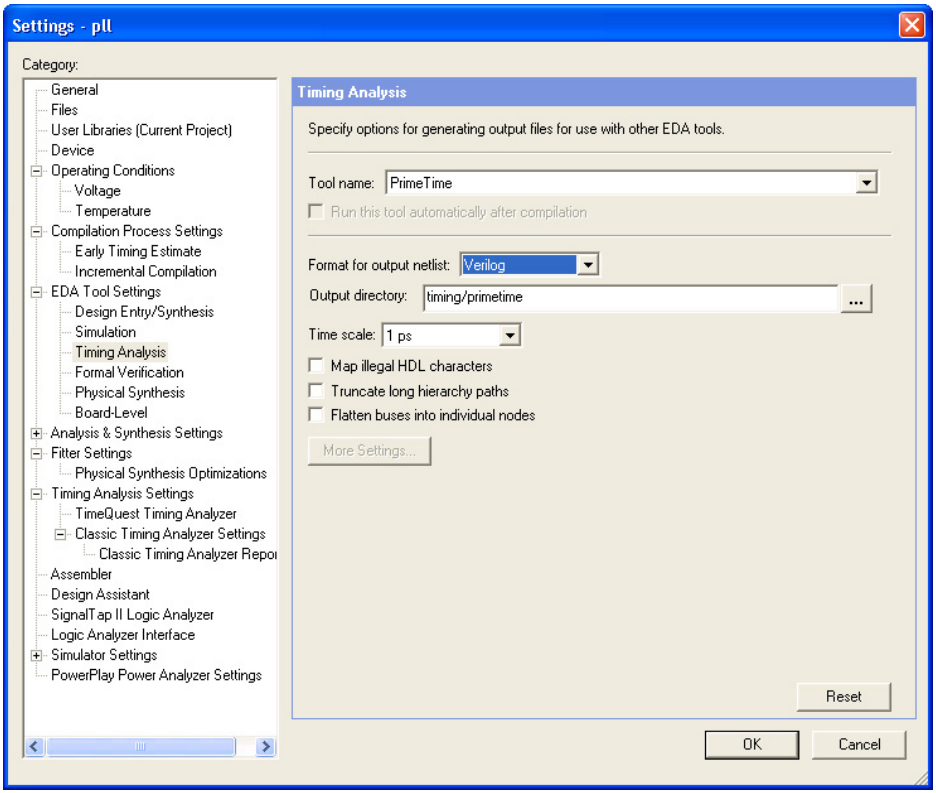
- "Quartus II Settings for Generating the PrimeTime Software Files" on page 9–2
- "Files Generated for the PrimeTime Software Environment" on page 9–3
- "Running the PrimeTime Software" on page 9–10
- "PrimeTime Timing Reports" on page 9–12
- "Static Timing Analyzer Differences" on page 9–23

Quartus II Settings for Generating the PrimeTime Software Files

To set the Quartus II software to generate files for the PrimeTime software, perform the following steps:

1. In the Quartus II software, on the Assignments menu, click **EDA Tool Settings**.
2. In the **Category** list, under **EDA Tool Settings,** select **Timing Analysis**.
3. In the **Tool name** drop-down list, select **PrimeTime**, and in the **Format for output netlist** drop-down list, select either **Verilog** or **VHDL**, depending on the HDL language you chose for use with the PrimeTime software (Figure 9–2).

Figure 9–2. Setting the Quartus II Software to Generate the PrimeTime Software Files



When you compile your project after making these settings, the Quartus II software runs the EDA Netlist Writer to create three files for the PrimeTime software. These files are saved in the <*revision_name*>**/timing/primetime directory** by default, where <*revision_name*> is the name of your Quartus II software revision. If it is not, you have used the wrong variable name.

Files Generated for the PrimeTime Software Environment

The Quartus II software generates a flattened netlist, a Standard Delay Output File (**.sdo**), and a Tcl script that prepares the PrimeTime software for timing analysis of the Quartus II project. These files are saved in the <*project directory*>**/timing/primetime** directory.

The Quartus II software uses the EDA Netlist Writer to generate PrimeTime files based on either the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer static timing analysis results. When you run the EDA Netlist Writer, the PrimeTime SDO files are based on delays generated by the currently selected timing analysis tool in the Quartus II software.

To specify the timing analyzer, on the Assignments menu, click **Settings**. The **Settings** dialog box appears. Under **Category**, click **Timing Analysis Settings**. Select the timing analyzer of your choice.



For more information about specifying the Quartus II timing analyzers, refer to either the *Quartus II Classic Timing Analyzer* or the *Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*. Also, refer to the *Switching to the* Quartus II *TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* to help you decide which timing analyzer is most appropriate for your design.

The Netlist

Depending on whether **Verilog** or **VHDL** is selected as the **Format for output netlist** option, in the **Tool name** list on the **Timing Analysis** page of the **Settings** dialog box, the netlist is written and saved as either <*project name*>.**vo** or <*project name*>.**vho**, respectively. This file contains the flattened netlist representing the entire design.



When the Quartus II TimeQuest Timing Analyzer is selected, only a Verilog PrimeTime netlist is generated.

The SDO File

The Quartus II software saves the Standard Delay Format Output (**.sdo**) File as either <*revision_name*>**_v.sdo** or <*revision_name*>**_vhd.sdo**, depending on whether you selected **Verilog** or **VHDL** in the **Tool name** list on the **Timing Analysis** page of the **Settings** dialog box.

This file contains the timing information for each timing path between any two nodes in the design.

When the Quartus II Classic Timing Analyzer is enabled, the slow-corner (worst case) timing models are used by default when generating the SDO file. To generate the SDO file using the fast-corner (best case) timing models, perform the following steps:

1. In the Quartus II software, on the Processing menu, point to Start and click **Start Classic Timing Analyzer (Fast Timing Model)**.

2. After the fast-corner timing analysis is complete, on the Processing menu, point to Start and click **Start EDA Netlist Writer** to create a <*revision_name*>**_v_fast.sdo** or <*revision_name*>**_vhd_fast.sdo** file, which contains the best-case delay values for each timing path.



If you are running a best-case timing analysis, the Quartus II software generates a Tcl script similar to the following: <*revision_name*>**_pt_v_fast.tcl**.

When TimeQuest is run with the fast-corner netlist or when the **Optimize fast-corner timing** check box is selected in the **Fitter Settings** dialog box, the fast-corner SDC file is generated.

After the EDA Netlist Writer has finished, two SDO files are created: <*revision_name*>**_v.sdo** (slow-corner) or <*revision_name*>**_v_fast.sdo** (fast-corner).

Generating Multiple Operating Conditions with TimeQuest

Different operating conditions can be specified to the EDA Netlist Writer for PrimeTime analysis. The different operating conditions are reflected in the .**sdo** file generated by the EDA Netlist Writer.

Table 9–1 shows the available operating conditions that can be set for a few of Altera's device families.

Table 9–1. Available Operating Condition Combinations

Device Family	Available Conditions (Model, Voltage, Temperature)
Stratix III	(slow, 1100 mV, 85° C), (slow, 1100 mV, 0° C), (fast, 1100 mV, 0° C)
Cyclone III	(slow, 1200 mV, 85° C), (slow, 1200 mV, 0° C), (fast, 1200 mV, 0° C)
Stratix II	(slow, N/A, N/A), (fast, N/A, N/A)
Cyclone II	(slow, N/A, N/A), (fast, N/A, N/A)



From the TimeQuest Console pane, use the command `get_available_operating_conditions` to obtain a list of available operating conditions for the target device.

The following steps shows how to generate the .**sdo** files for the three different operating conditions for a Stratix III design. Each command must be entered at the command prompt.



The –tq2pt option for `quartus_sta` is required only if the project doesn't specify that PrimeTime tool will be used as the timing analysis tool.

1. Generate the first slow corner model at the operating conditions: slow, 1100 mV, and 85° C.

```
quartus_sta --model=slow --voltage=1100 --
temperature=85 <project name>
```

2. Generate the fast corner model at the operating conditions: fast, 1100 mV, and 0° C.

```
quartus_sta --model=fast --voltage=1100 --
temperature=0 --tq2pt <project name>
```

3. Generate the PrimeTime output files for the corners specified above. The output files will be generated in the primetime_two_corner_files directory.

```
quartus_eda --timing_analysis --tool=primetime
--format=verilog --
output_directory=primetime_two_corner_files --
write_settings_files=off <project name>
```

4. Generate the second slow corner model at the operating conditions: slow, 1100 mV, and 0º C.

```
quartus_sta --model=slow --voltage=1100 --
temperature=0 --tq2pt <project name>
```

5. Generate the PrimeTime output files for the second slow corner. The output files will be generated in the primetime_one_slow_corner_files directory.

```
quartus_eda --timing_analysis --tool=primetime --
format=verilog --
output_directory=primetime_one_slow_corner_files -
-write_settings_files=off $revision
```

To summarize, the previous steps generate the following files for the three operating conditions:

- First slow corner (slow, 1100 mV, 85º C) :
  VO File—primetime_two_corner_files/<*project name*>.**vo**
  SDO File—primetime_two_corner_files/<*project name*>_v.**sdo**

- Fast corner (fast, 1100 mV, 0º C) :
  VO File—primetime_two_corner_files/<*project name*>.vo
  SDO File—primetime_two_corner_files/<*project name*>_v_fast.**sdo**

- Second slow corner (slow, 1100 mV, 0º C) :
  VO File—primetime_one_slow_corner_files/<*project name*>.vo
  SDO File—primetime_one_slow_corner_files/<*project name*>_v.**sdo**



The directory primetime_one_slow_corner_files may also have files for fast corner. These files can be ignored since they were already generated in the primetime_two_corner_files directory.

The Tcl Script

The Tcl script generated by the Quartus II software contains information required by the PrimeTime software to analyze the timing and set up your post-fit design. This script specifies the search path and the names of the PrimeTime database library files provided with the Quartus II software. The `search_path` and `link_path` variables are defined at the beginning of the Tcl file. The `link_path` variable is a space-delimited list that contains the names of all database files used by the PrimeTime software.

Depending on whether you selected **Verilog** or **VHDL** in the **Format for output netlist** list on the **Timing Analysis** page of the **Settings** dialog box, when the Quartus II Classic Timing Analyzer is enabled, the EDA Netlist Writer generates and saves the script as either <*revision_name*>**_pt_v.tcl** or <*revision_name*>**_pt_vhd.tcl**.

To access the **EDA Settings** dialog box, on the Assignments menu, click **EDA Tool Settings**, then expand **EDA Tool Settings** under the Category list. In the dialog box, you can specify VHDL or Verilog for the format for the output netlist.



The script also directs the PrimeTime software to use the <*device family*>**_all_pt.v** or <*device family*>**_all_pt.vhd** file, which contains the Verilog or VHDL description of library cells for the targeted device family.

Example 9–1 shows the `search_path` and `link_path` variables defined in the Tcl script:

Example 9–1. Sample PrimeTime Setup Script

```
set quartus_root "altera/quartus/"
set search_path [list . [format "%s%s" $quartus_root "eda/synopsys/primetime/lib"]  ]

set link_path [list *  stratixii_lcell_comb_lib.db  stratixii_lcell_ff_lib.db
stratixii_asynch_io_lib.db  stratixii_io_register_lib.db  stratixii_termination_lib.db
bb2_lib.db  stratixii_ram_internal_lib.db  stratixii_memory_register_lib.db
stratixii_memory_addr_register_lib.db  stratixii_mac_out_internal_lib.db
stratixii_mac_mult_internal_lib.db  stratixii_mac_register_lib.db
stratixii_lvds_receiver_lib.db stratixii_lvds_transmitter_lib.db
stratixii_asmiblock_lib.db stratixii_crcblock_lib.db  stratixii_jtag_lib.db
stratixii_rublock_lib.db  stratixii_pll_lib.db  stratixii_dll_lib.db alt_vtl.db]

read_vhdl  -vhdl_compiler  stratixii_all_pt.vhd
```

The EDA Netlist Writer converts any Quartus II Classic Timing Analyzer timing assignments to the PrimeTime software constraints and exceptions when it generates the PrimeTime files. The converted constraints are saved to the Tcl script. The Tcl script also includes a

PrimeTime software command that reads the Standard Delay Format Output (**.sdo**) file generated by the Quartus II software. You can place additional commands in the Tcl script to analyze or report on timing paths.

Table 9–2 shows some examples of timing assignments converted by the Quartus II software for the PrimeTime software. For example, the `set_input_delay -max` command sets the input delay on an input pin.

Table 9–2. Equivalent Quartus II and PrimeTime Software Constraints

Quartus II Equivalent	PrimeTime Constraint
Clock defined on input pin, clock of 10 ns period and 50% duty cycle	`create_clock -period 10.000 -waveform {0 5.000} \ [get_ports clk] -name clk`
Input maximum delay of 1 ns on input pin, **din**	`set_input_delay -max -add_delay 1.000 -clock \ [get_clocks clk] [get_ports din]`
Input minimum delay of 1 ns on input pin, **din**	`set_input_delay -min -add_delay 1.000 -clock \ [get_clocks clk] [get_ports din]`
Output maximum delay of 3 ns on output pin, **out**	`set_output_delay -max -add_delay 3.000 -clock \ [get_clocks clk] [get_ports out]`

When the Quartus II TimeQuest Timing Analyzer is turned on, the EDA Netlist Writer generates and saves the script as <*revision_name*>**.pt.tcl**.

The EDA Netlist Writer converts all Quartus II TimeQuest Timing Analyzer SDC constraints and exceptions into compatible PrimeTime software constraints and exceptions when it generates the PrimeTime files. The constraints and exceptions are saved to the <*revision_name*>**.constraints.sdc** file.

Generated File Summary

The files that are generated by the EDA Netlist Writer for the PrimeTime software depend on the Quartus II timing analysis tool you selected.

Table 9–3 shows the files that are generated for the PrimeTime software when the Quartus II Classic Timing Analyzer is selected.

Table 9–3. Quartus II Classic Timing Analyzer-Generated PrimeTime Files

File	Description
<*revision_name*>**.vho** | <*revision_name*>**.vo**	The PrimeTime software output netlist. Either a VHDL Output File or a Verilog Output file is generated, depending on the output netlist language set.
<*revision_name*>**_vhd.sdo** | <*revision_name*>**_v.sdo**	The PrimeTime software standard delay file. Either a VHDL Standard Delay Output file or a Verilog Standard Delay Output file is generated, depending on the output netlist language set.
<*revision_name*>**_pt_vhd.tcl** | <*revision_name*>**_pt_v.tcl**	PrimeTime setup and constraint script. Either a VHDL Tcl script or a Verilog Tcl script is generated, depending on the output netlist language set.

Table 9–4 shows the files that are generated for the PrimeTime software when the Quartus II TimeQuest Timing Analyzer is selected. The EDA Netlist Writer supports the output netlist format only when the TimeQuest Timing Analyzer is enabled.

Table 9–4. Quartus II TimeQuest Timing Analyzer-Generated PrimeTime Files

File	Description
<*revision_name*>**.vo**	The PrimeTime software output netlist. When the Quartus II TimeQuest Timing Analyzer is enabled, only PrimeTime (Verilog) is supported.
<*revision_name*>**_v.sdo** | <*revision_name*>**_v_fast.sdo**	The PrimeTime software standard delay file. When the Quartus II TimeQuest Timing Analyzer is enabled, only PrimeTime (Verilog) is supported.
<*revision_name*>**.pt.tcl**	PrimeTime setup and constraint script. When the Quartus II TimeQuest Timing Analyzer is enabled, only PrimeTime (Verilog) is supported.
<*revision_name*>**.collections.sdc**	Contains the mapping from the Quartus II TimeQuest Timing Analyzer netlist to the PrimeTime netlist.
<*revision_name*>**.constraints.sdc**	Contains the converted Quartus II TimeQuest Timing Analyzer constraints for the PrimeTime software.

Running the PrimeTime Software

The PrimeTime software runs only on UNIX operating systems. If the Quartus II output files for the PrimeTime software were generated by running the Quartus II software on a PC/Windows-based system, follow these steps to run the PrimeTime software using Quartus II output files:

1. Install the PrimeTime libraries on a UNIX system by installing Quartus II software on UNIX.

   The PrimeTime libraries are located in the <*Quartus II installation directory*>**/eda/synopsys/primetime/lib** directory.

2. Copy the Quartus II output files to the appropriate UNIX directory. You may need to run a PC to UNIX program, such as `dos2unix`, to remove any control characters.

3. Modify the Quartus II path in Tcl scripts to point to the PrimeTime libraries, as described in Step 1. In Example 9–1, the first line is:

   set quartus_root "altera/quartus/" set search_path [list . [format "%s%s" $quartus_root "eda/synopsys/primetime/lib"]]

   This is the Tcl script that should be modified.

Analyzing Quartus II Projects

The PrimeTime software is controlled with Tcl scripts and can be run through `pt_shell`. You can run the <*revision_name*>**_pt_v.tcl** script file. For example, type the following at a UNIX system command prompt:

```
pt_shell -f <revision_name>_pt_v.tcl ↵
```

When the Quartus II TimeQuest Timing Analyzer is selected, type the following at a UNIX system command prompt:

```
pt_shell -f <revision_name>.pt.tcl ↵
```

After all Tcl commands in the script are interpreted, the PrimeTime software returns control to the `pt_shell` prompt, which allows you to use other commands.

Other pt_shell Commands

You can run additional pt_shell commands at the pt_shell prompt, including the man program. For example, to read documentation about the report_timing command, type the following at the pt_shell prompt:

```
man report_timing ↵
```

You can list all commands available in pt_shell by typing the following at the pt_shell prompt:

```
help ↵
```

Type quit ↵ at the pt_shell prompt to close pt_shell.



You can also run pt_shell without a script file by typing pt_shell↵ at the UNIX command line prompt.

PrimeTime Timing Reports

Sample of the PrimeTime Software Timing Report

After running the script, the PrimeTime software generates a timing report. If the timing constraints are not met, `Violated` is displayed at the end of the timing report. The timing report also gives the negative slack.

The PrimeTime software report is similar to the sample shown in Example 9–2. The starting point in this report is a register clocked by clock signal, `clock`, the endpoint is another register, `inst3-I.lereg`.

Example 9–2. Hold Path Report in PrimeTime

```
Startpoint: inst2~I.lereg
  (rising edge-triggered flip-flop clocked by clock)
Endpoint: inst3~I.lereg
  (rising edge-triggered flip-flop clocked by clock)
Path Group: clock
Path Type: min
Point  IncrPath
-------------------------------------------------------------
clock clock (rise edge)0.0000.000
clock network delay (propagated)3.1663.166
inst2~I.lereg.clk (stratix_lcell_register)0.000 3.166r
inst2~I.lereg.regout (stratix_lcell_register)  <-0.176*3.342r
inst2~I.regout (stratix_lcell)0.000*3.342r
inst3~I.datac (stratix_lcell)0.000*3.342r
inst3~I.lereg.datac (stratix_lcell_register)3.413*6.755r
data arrival time6.755
clock clock (rise edge)0.0000.000
clock network delay (propagated)3.0023.002
inst3~I.lereg.clk (stratix_lcell_register)3.002r
library hold time0.100*3.102
data required time 3.102
-------------------------------------------------------------
data required time3.102
data arrival time-6.755
-------------------------------------------------------------
slack (MET)3.653
```

Comparing Timing Reports from the Quartus II Classic Timing Analyzer and the PrimeTime Software

Both the Quartus II Classic Timing Analyzer and the Quartus II TimeQuest Timing Analyzer generate a static timing analysis report for every successful design compilation. The timing report lists all of the analyzed timing paths in your design that were analyzed, and indicates whether these paths have met or violated their timing requirements. Violations are reported only if timing constraints were specified.

The Quartus II TimeQuest Timing Analyzer uses an equivalent set of equations as PrimeTime when reporting the static timing analysis result for a design. However, the Quartus II Classic Timing Analyzer uses slightly different reporting equations when reporting the static timing analysis results for a design. This section describes these differences between the Quartus II Classic Timing Analyzer and the PrimeTime software.

The timing report generated by the Quartus II Classic Timing Analyzer differs from the report generated by the PrimeTime software. Both tools provide the same data but present in different formats. The following sections show how the PrimeTime software reports the following slack values differently from the Quartus II Classic Timing Analyzer report:

- "Clock Setup Relationship and Slack" on page 9–13
- "Clock Hold Relationship and Slack" on page 9–17
- "Input Delay and Output Delay Relationships and Slack" on page 9–21

Clock Setup Relationship and Slack

The Quartus II Classic Timing Analyzer performs a setup check that ensures that the data launched by source registers is latched correctly at the destination registers. The Quartus II Classic Timing Analyzer does this by determining the data arrival time and clock arrival time at the destination registers, and compares this data with the setup time delay of the destination register. Equation 1 expresses the inequality that is used for a setup check. The data arrival time includes the longest path from the clock to the source register, the clock-to-out micro delay of the source register, and the longest path from the source register to the destination register. The clock arrival time is the shortest delay from the clock to the destination register.

$$\text{Clock Arrival} - \text{Data Arrival} \geq t_{su} \quad (1)$$

Slack is the margin by which a timing requirement is met or not met. Positive slack indicates the margin by which a requirement is met. Negative slack indicates the margin by which a requirement was not met. The Quartus II Classic Timing Analyzer determines the clock setup slack, with Equation 2:

(2) Clock Setup Slack = Largest Register-to-Register Requirement – Longest Register-to-Register Delay



The longest register-to-register delay in the previous equation is equal to the register-to-register data delay.

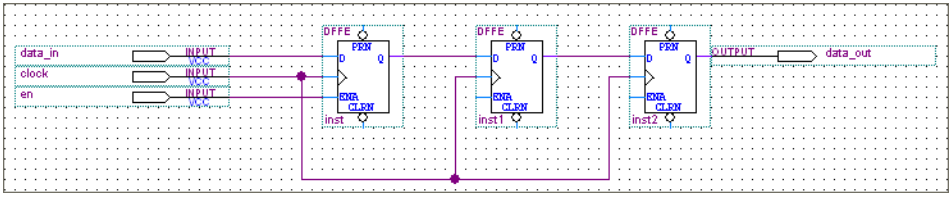
(3) Largest Register-to-Register Requirement = Setup Relationship between Source and Destination + Largest Clock Skew – Micro t_{co} of Destination Register – Micro t_{su} of Destination Register

Setup Relationship between Source and Destination = Latch Edge – Launch Edge

Clock Skew = Shortest Clock Path to Destination – Longest Clock Path to Source

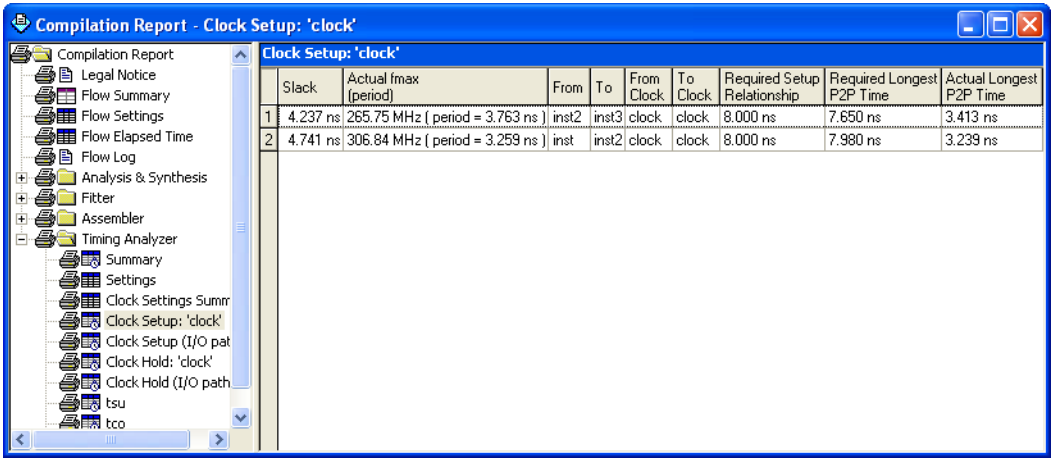
For a simple three-register design, refer to Figure 9–3.

Figure 9–3. Simple Three-Register Design



The Quartus II Classic Timing Analyzer generates a report for the design, as shown in Figure 9–4.

Figure 9–4. Timing Analyzer Report from Figure 9–3



Equation 1, 2, and 3 are similar to those found in other static timing analysis tools, such as the PrimeTime software. Equation 4, 5, 6, and 7, used by the PrimeTime software, are essentially the same as those used by the Quartus II Classic Timing Analyzer, but they are rearranged.

(4) Slack = Data Required – Data Arrival

(5) Clock Arrival = Latch Edge + Shortest Clock Path to Destination

(6) Data Required = Clock Arrival – Micro t_{su}

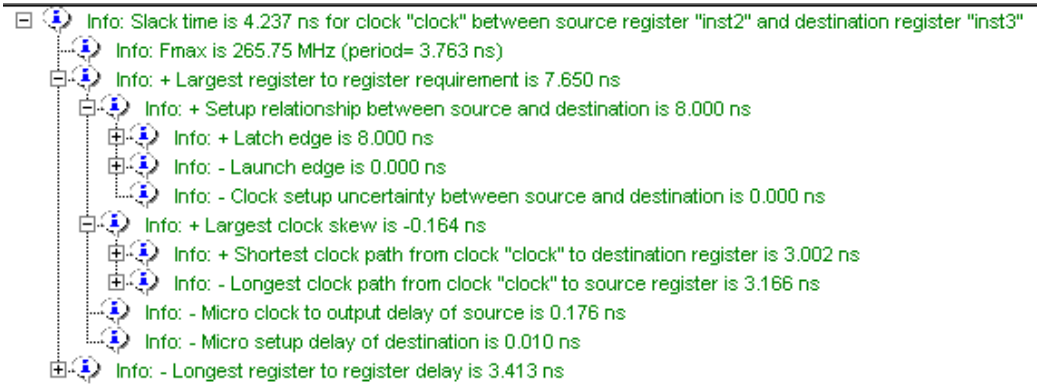
(7) Data Arrival = Launch Edge + Longest Clock Path to Source + Micro t_{co} + Longest Data Delay



The longest data delay in the previous equation is equal to register-to-register data delay.

Figure 9–5 shows a clock setup check in the Quartus II software.

Figure 9–5. Clock Setup Check Reporting with the Quartus II Classic Timing Analyzer



The following results are obtained by extracting the numbers from the Quartus II Classic Timing Analyzer report and applying them to the clock setup slack equations from the Quartus II Classic Timing Analyzer:

(8)

Setup Relationship between Source and Destination = Latch Edge – Launch Edge – Clock Setup Uncertainty

$$8.0 - 0.0 - 0.0 = 8.0\,\text{ns}$$

Clock Skew = Shortest Clock Path to Destination – Longest Clock Path to Source

$$3.002 - 3.166 = -0.164\,\text{ns}$$

Largest Register-to-Register Requirement =
Setup Relationship between Source & Destination + Largest Clock Skew
– Micro t_{co} of Source Register – Micro t_{su} of Destination Register

$$8 + (-0.164) - 0.176 - 0.010 = 7.650\,\text{ns}$$

Clock Setup Slack = Largest Register-to-Register Requirement – Longest Register-to-Register Delay

$$7.650 - 3.413 = 4.237\,\text{ns}$$

For the same register-to-register path, the PrimeTime software generates a clock setup report as shown in Example 9–3:

Example 9–3. Setup Path Report in PrimeTime

```
Startpoint: inst2~I.lereg
  (rising edge-triggered flip-flop clocked by clock)
Endpoint: inst3~I.lereg
  (rising edge-triggered flip-flop clocked by clock)
Path Group: clock
Path Type: max  PointIncrPath
-------------------------------------------------------------
clock clock (rise edge)0.0000.000
clock network delay (propagated)3.1663.166
inst2~I.lereg.clk (stratix_lcell_register)0.0003.166r
inst2~I.lereg.regout (stratix_lcell_register) <-0.176*3.342r
inst2~I.regout (stratix_lcell) <- 0.000*3.342r
inst3~I.datac (stratix_lcell) <-0.000*3.342r
inst3~I.lereg.datac (stratix_lcell_register)3.413*6.755r
data arrival time6.755
clock clock (rise edge)8.0008.000
clock network delay (propagated)3.00211.002
inst3~I.lereg.clk (stratix_lcell_register)11.002r
library setup time-0.010*10.992
data required time10.992
-------------------------------------------------------------
data required time10.992
data arrival time-6.755
-------------------------------------------------------------
slack (MET)4.237
```

Clock Hold Relationship and Slack

The Quartus II Classic Timing Analyzer performs a hold time check along every register-to-register path in the design to ensure that no hold time violations have occurred. The hold time check verifies that data from the source register does not reach the destination until after the hold time of the destination register. The condition used for a hold check is shown in Equation 9:

$$\text{Data Arrival} - \text{Clock Arrival} \geq t_H \quad (9)$$

The Quartus II Classic Timing Analyzer determines the clock hold slack with Equation 10, 11, 12, and 13:

$$\text{Clock Hold Slack} = \text{Shortest Register-to-Register Delay} - \text{Smallest Register-to-Register Requirement} \quad (10)$$

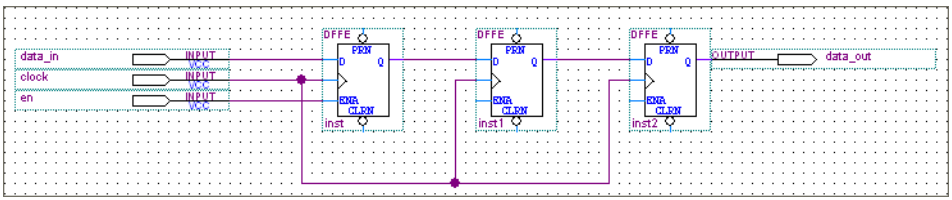
(11) Smallest Register-to-Register Requirement = Hold Relationship between Source & Destination + Smallest Clock Skew – Micro t_{su} of Source + Micro t_H of Destination

(12) Hold Relationship between Source & Destination = Latch Edge – Launch Edge

(13) Smallest Clock Skew = Longest Clock Path from Clock to Destination Register – Shortest Clock Path from Clock to Source Register

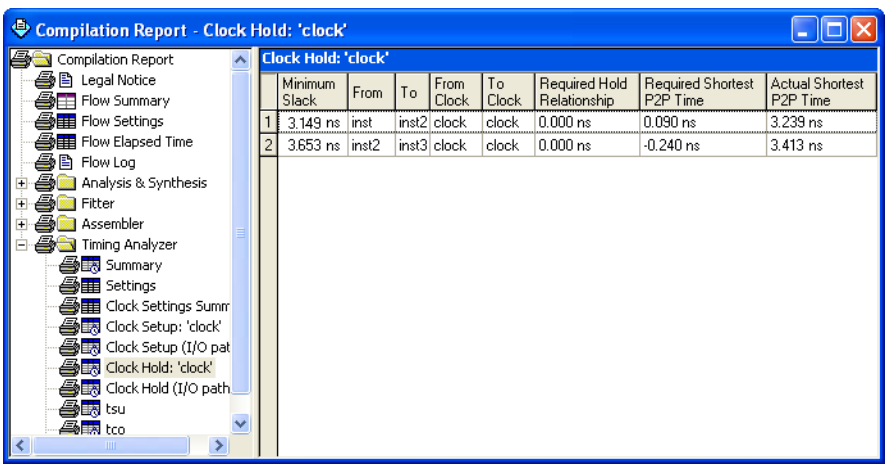
Figure 9–6 shows a simple three-register design.

Figure 9–6. A Simple Three-Register Design



The Quartus II Classic Timing Analyzer generates a report as shown in Figure 9–7.

Figure 9–7. Timing Analyzer Report Generated from the Three Register Design



The previous equations are similar to those found in the Quartus II software. The following equations are the same equations that are used by the PrimeTime software, but they are rearranged.

(14) Slack = Data Required – Data Arrival

(15) Clock Arrival = Latch Edge + Longest Clock Path to Destination

(16) Data Required = Clock Arrival – Micro t_H

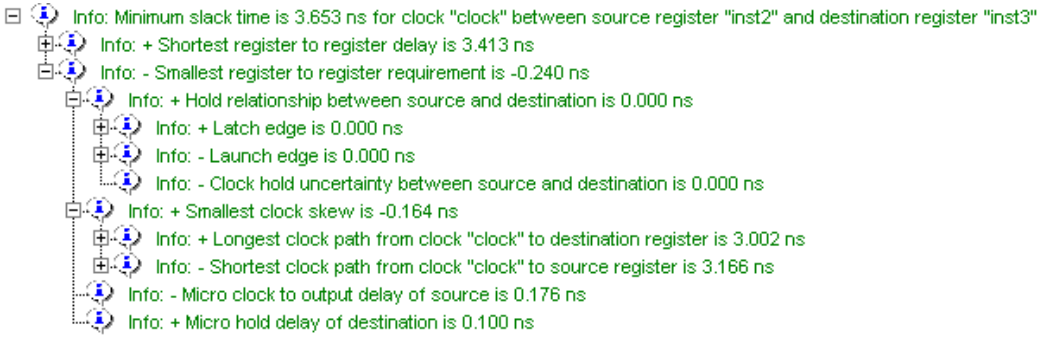
(17) Data Arrival = Launch Edge + Longest Clock Path to Source + Micro t_{co} + Shortest Data Delay



The shortest register-to-register delay in the previous equation is equal to register-to-register data delay.

Figure 9–8 shows a clock setup check with the Quartus II Classic Timing Analyzer.

Figure 9–8. Clock Hold Check Reporting with the Quartus II Classic Timing Analyzer



The following results are obtained by extracting the numbers from the Timing Analysis report and applying the clock setup slack equations from the Quartus II Classic Timing Analyzer.

(18) Clock Hold Slack = Shortest Register-to-Register Delay – Smallest Register-to-Register Requirement

$$3.413 - (-0.240) = 3.653\,\text{ns}$$

Smallest Register-to-Register Requirement = Hold Relationship between Source & Destination + Smallest Clock Skew – Micro t_{co} of Source + Micro t_H of Destination

$$0 + (-0.164) - 0.176 + 0.100 = -0.240\,\text{ns}$$

Hold Relationship between Source & Destination = Latch – Launch

0.0 – 0.0 ns

Smallest Clock Skew = Longest Clock Path from Clock to Destination Register – Shortest Clock Path from Clock to Source Register

3.002 – 3.166 = –0.164 ns

For the same register-to-register path, the PrimeTime software generates the report shown in Example 9–4:

Example 9–4. Hold Path Report in PrimeTime

```
Startpoint: inst2~I.lereg
  (rising edge-triggered flip-flop clocked by clock)
Endpoint: inst3~I.lereg
  (rising edge-triggered flip-flop clocked by clock)
Path Group: clock
Path Type: min
Point  IncrPath
    ------------------------------------------------------------
  clock clock (rise edge)0.0000.000
  clock network delay (propagated)3.1663.166
  inst2~I.lereg.clk (stratix_lcell_register)0.0003.166r
  inst2~I.lereg.regout (stratix_lcell_register)<-0.176*3.342r
  inst2~I.regout (stratix_lcell)0.000*3.342r
  inst3~I.datac (stratix_lcell)0.000*3.342r
  inst3~I.lereg.datac (stratix_lcell_register)3.413*6.755r
  data arrival time6.755

  clock clock (rise edge)0.0000.000
  clock network delay (propagated)3.0023.002
  inst3~I.lereg.clk (stratix_lcell_register)3.002r
  library hold time0.100*3.102
  data required time                                3.102
    ------------------------------------------------------------
  data required time3.102
  data arrival time-6.755
    ------------------------------------------------------------
  slack (MET)3.653
```

Both sets of hold slack equations can be used to determine the hold slack value of any path.

Input Delay and Output Delay Relationships and Slack

Input delay and output delay reports generated by the Quartus II Classic Timing Analyzer are similar to the clock setup and clock hold relationship reports. Figure 9–9 shows the input delay and output delay report for the design shown in Figure 9–6 on page 9–18.

Figure 9–9. Input and Output Delay Reporting with the Quartus II Classic Timing Analyzer

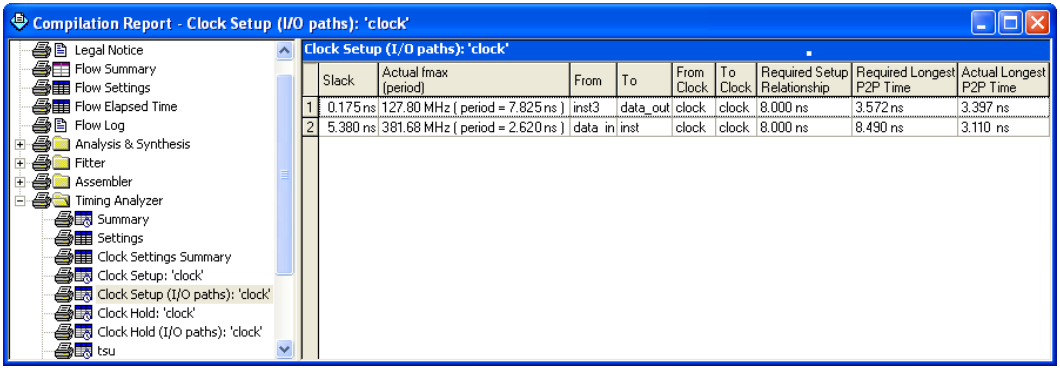
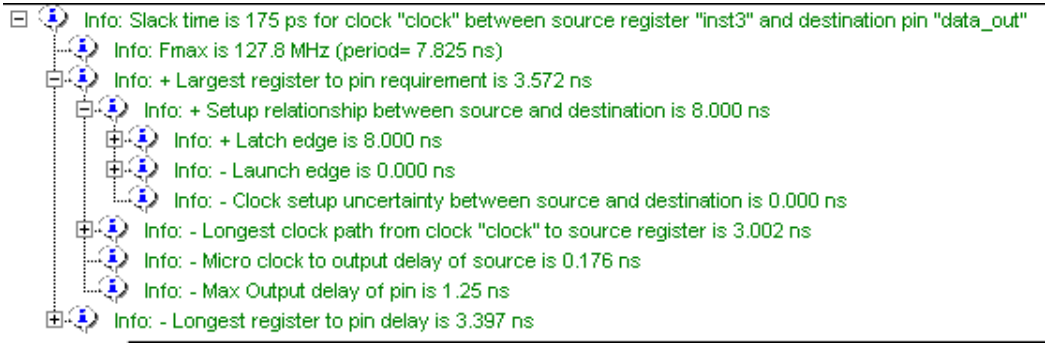


Figure 9–10 shows the fully expanded view for the output delay path.

Figure 9–10. Output Delay Path Reporting with the Quartus II Classic Timing Analyzer



For the same output delay path, the PrimeTime software generates a report similar to Example 9–5:

Example 9–5. Setup Path Report in PrimeTime

```
Startpoint: inst3~I.lereg
   (rising edge-triggered flip-flop clocked by clock)
Endpoint: data_out
   (output port clocked by clock)
Path Group: clock
Path Type: max  PointIncrPath
-----------------------------------------------------------------
clock clock (rise edge)0.0000.000
clock network delay (propagated)3.0023.002
inst3~I.lereg.clk (stratix_lcell_register)0.0003.002r
inst3~I.lereg.regout (stratix_lcell_register)<- 0.176*3.178r
inst3~I.regout (stratix_lcell)<- 0.0003.178r
data_out~I.datain (stratix_io)<- 0.000 3.178r
data_out~I.out_mux3.A (mux21) <-0.0003.178r
data_out~I.out_mux3.MO (mux21)<- 0.000  3.178r
data_out~I.and2_22.IN1 (AND2)<- 0.0003.178r
data_out~I.and2_22.Y (AND2)<- 0.0003.178r
data_out~I.out_mux1.A (mux21)<-0.0003.178r
data_out~I.out_mux1.MO (mux21)<- 0.0003.178r
data_out~I.inst1.datain (stratix_asynch_io)<-0.902*4.080r
data_out~I.inst1.padio (stratix_asynch_io)<- 2.495*6.575r
data_out~I.padio (stratix_io)<- 0.000 6.575r
data_out (out)0.0006.575r
data arrival time6.575
clock clock (rise edge)8.0008.000
clock network delay (propagated)0.0008.000
output external delay1.2506.750
data required time6.750
-----------------------------------------------------------------
data required time6.750
data arrival time6.575
-----------------------------------------------------------------
slack (MET)   0.175
```

To generate a list of the 100 worst paths and place this data into a file called **file.timing**, type the following command at the `pt_shell` prompt:

```
report_timing -nworst 100 > file.timing ↵
```

Timing paths in the PrimeTime software are listed in the order of most-negative-slack to most-positive-slack. The PrimeTime software does not categorize failing paths by default. Timing setup (t_{SU}) and timing hold (t_H) times are not listed separately. In the PrimeTime software, each path is shown with a start and end point; for example, if it is a

register-to-register or input-to-register type of path. If you only use the `report_timing` part of the command without adding a `-delay` option, only the `setup-time-related` timing paths are reported.

The following command is used to create a minimum timing report or a list of hold-time-related violations:

```
report_timing -delay_type min ↵
```

Ensure that the correct SDO file, either minimum or maximum delays, is loaded before running this command.

Static Timing Analyzer Differences

Under certain design conditions, several static timing analysis differences can exist between the Classic Timing Analyzer and the TimeQuest Timing Analyzer, and the PrimeTime software. The following sections explain the differences between the two static timing analysis engines and the PrimeTime software.

The Quartus II Classic Timing Analyzer and the PrimeTime Software

The following section describes the differences between the Quartus II Classic Timing Analyzer and the PrimeTime software.

Rise/Fall Support

The Quartus II Classic Timing Analyzer does not support rise/fall analysis. However, rise/fall support is available in PrimeTime.

Minimum and Maximum Delays

TimeQuest calculates minimum and maximum delays for all device components with the exception of clock routing. PrimeTime does not model these delays. This can result in different slacks for a given path on average by 2 - 3%.

Recovery/Removal Analysis

TimeQuest performs a more pessimistic recovery/removal analysis for asynchronous path than PrimeTime. This can result in different delays reported between the two tools.

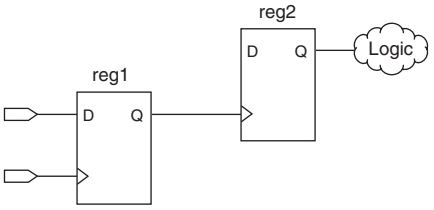
Encrypted Intellectual Property Blocks

The Quartus II software has the capability to decrypt all intellectual property (IP) blocks designed for Altera® devices that have been encrypted by their vendors. The decryption process allows the Quartus II software to perform a full compilation of the design that contains an encrypted IP block. This also allows the Quartus II Classic Timing Analyzer to perform a complete static timing analysis on the design. However, when the PrimeTime software is designated as the static timing analysis tool, the Quartus II EDA Netlist Writer does not generate either a VHDL Output File (**.vho**) or Verilog Output File (**.vo**) netlist file for designs that contain encrypted IP blocks for which the license does not permit generation of output netlists for third-party tools.

Registered Clock Signals

Registered clock signals are clock signals that pass through a register before reaching the clock port of a sequential element. Figure 9–11 shows an example of a registered clock signal.

Figure 9–11. Registered Clock Signal



If no clock setting is applied to the register on the clock path (shown as register `reg_1` in Figure 9–11), the Quartus II Classic Timing Analyzer treats the register in the clock path as a buffer. The delay of the buffer is equal to the CELL delay of the register plus the t_{CO} of the register. The PrimeTime software does not treat the register as a buffer.

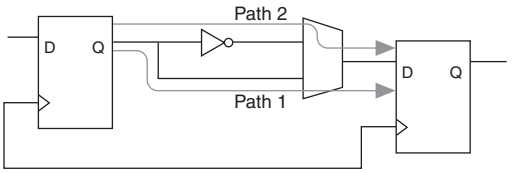


For more information about creating clock settings, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Multiple Source and Destination Register Pairs

In any design, multiple paths may exist from a source register to a destination register. Each path from the source register to the destination register may have a different delay value due to the different routes taken. For example, Figure 9–12 shows a sample design that contains multiple path pairs between the source register and destination register.

Figure 9–12. Multiple Source and Destination Pairs



The Quartus II Classic Timing Analyzer analyzes all source and destination pairs, but reports only the source and destination register pair with the worst slack. For example, if the Path 2 pair delay is greater than the Path 1 pair delay in Figure 9–12, the Quartus II Classic Timing Analyzer reports the slack value of the Path 2 pair and not the Path 1 pair. The PrimeTime software reports all possible source and destination register pairs.

Latches

By default, the Quartus II software implements all latches as combinational loops. The Quartus II Classic Timing Analyzer can analyze such latches by treating them as registers with inverted clocks or analyze latches as a combinational loop modeled as a combinational delay.



For more information about latch analysis, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

The PrimeTime software always analyzes these latches as combinational loops, as defined in the netlist file.

LVDS I/O

When it analyzes the dedicated LVDS transceivers in your design, the Quartus II Classic Timing Analyzer generates the Receiver Skew Margin (RSKM) report and a Channel-to-Channel Skew (TCCS) report. The PrimeTime software does not generate these reports.

Clock Latency

When a single clock signal feeds both the source and destination registers of a register-to-register path, and either an Early Clock Latency or a Late Clock Latency assignment has been applied to the clock signal, the Quartus II Classic Timing Analyzer does not factor in the clock latency values when it calculates the clock skew between the two registers. The Quartus II Classic Timing Analyzer factors in the clock latency values when the clock signal to the source and destination registers of a register-to-register path are different. The PrimeTime software applies the clock latency values when a single clock signal or different clock signals feeds the source and destination registers of a register-to-register path.

Input and Output Delay Assignments

When a purely combinational (non-registered) path exists between an input pin and output pin of the Altera FPGA and both pins have been constrained with an input delay and an output delay assignment applied, respectively, the Quartus II Classic Timing Analyzer does not perform a clock setup or clock hold analysis. The PrimeTime software analyzes these paths.

Generated Clocks Derived from Generated Clocks

The Quartus II Classic Timing Analyzer does not support a generated clock derived from a generated clock. This situation might occur if a generated clock feeds the input clock pin of a PLL. The output clock of the PLL is a generated clock.

The Quartus II TimeQuest Timing Analyzer and the PrimeTime Software

The following sections describe the static timing analysis differences between the Quartus II TimeQuest Timing Analyzer and the PrimeTime software.

Encrypted Intellectual Property Blocks

The Quartus II software has the capability to decrypt all IP blocks, designed for Altera devices that have been encrypted by their vendors. The decryption process allows the Quartus II software to perform a full compilation on the design containing an encrypted IP block. This also allows the Quartus II TimeQuest Timing Analyzer to perform a complete static timing analysis on the design. However, when the PrimeTime software is designated as the static timing analysis tool, the Quartus II

EDA Netlist Writer does not generate **.vho** or **.vo** netlist files for designs that contain encrypted IP blocks whose license does not permit generation of output netlists for other tools.

Latches

By default, the Quartus II software implements all latches as combinational loops. The Quartus II TimeQuest Timing Analyzer can analyze such latches by treating them as registers with inverted clocks. The Quartus II TimeQuest Timing Analyzer analyzes latches as a combinational loop modeled as a combinational delay.



For more information about latch analysis, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

The PrimeTime software always analyzes these latches as combinational loops, as defined in the netlist file.

LVDS I/O

When it analyzes the dedicated LVDS transceivers in your design, the Quartus II TimeQuest Timing Analyzer generates a Receiver Skew Margin (RSKM) report and a Channel-to-Channel Skew (TCCS) report. The PrimeTime software does not generate these reports.

The Quartus II TimeQuest Timing Analyzer SDC File and PrimeTime Compatibility

Because of differences between node naming conventions with the netlist generated by the EDA Netlist Writer and the internal netlist used by the Quartus II software, SDC files generated for the Quartus II software or the Quartus II TimeQuest Timing Analyzer are not compatible with the PrimeTime software.

Run the EDA Netlist Writer to generate a compatible SDC file from the TimeQuest SDC file for the PrimeTime software. After the files have been generated, <*revision_name*>**.collections.sdc** and <*revision_name*>**.constraints.sdc**, both files can be read in by the PrimeTime software for compatibility of constraints between the Quartus II TimeQuest Timing Analyzer and the PrimeTime software.

Clock and Data Paths

If a timing path acts both as a clock path (a path that connects to a clock pin with a clock associated to it), and a data path (a path that feeds into the data in port of a register), the Quartus II TimeQuest Timing Analyzer will report the data paths, whereas PrimeTime will not.

Inverting and Non-Inverting Propagation

TimeQuest always propagates non-inverting sense for clocks through non-unate paths in the clock network.

PrimeTime's default behavior is to propagate both inverting and non-inverting senses through a non-unate path in the clock network.

Multiple Rise/Fall Numbers For a Timing Arc

For a given timing path with a corresponding set of pins/ports that make up the path (including source and destination pair), if the individual components of that path have different rise/fall delays, there can potentially be many timing paths with different delays using the same set of pins. If this occurs, TimeQuest reports only one timing path for the set of pins that make up the path.

Virtual Generated Clocks

PrimeTime does not support generated clocks that are virtual. To maintain compatibility between TimeQuest and PrimeTime, all generated clocks should have an explicit target specified.

Generated Clocks Derived from Generated Clocks

The Quartus II Classic Timing Analyzer does not support the creation of a generated clock derived from a generated clock. This situation might occur if a generated clock feeds the input clock pin of another generated clock. The output clock of the PLL is a generated clock.

Conclusion

The Quartus II software can export a netlist, constraints, and timing information for use with the PrimeTime software. The PrimeTime software can use data from either best-case or worst-case Quartus II timing models to measure timing. The PrimeTime software is controlled using a Tcl script generated by the Quartus II software that you can customize to direct the PrimeTime software to produce violation and slack reports.

Referenced Documents

This chapter references the following document:

- *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer chapter* in volume 3 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* in volume 3 of the *Quartus II Handbook*
- *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*

Document Revision History

Table 9–5 shows the revision history for this chapter.

Table 9–5. Document Revision History

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 9–29.	—
May 2007 v7.1.0	• Added Generating Multiple Operating Conditions with TimeQuest • Added Rise/Fall Support • Added Minimum and Maximum Delays • Added Recovery/Removal Aanalyis • Added Generated Clocks Derived from Generated Clocks • Added Multiple Rise/Fall Numbers for a Timing Analyzer SDC • Virtual Generated Clocks • Added Referenced Documents	Updates added to the Static Timing Analyzer Differences section of this chapter.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	• Noted the differences between the different timing analyzers • Explained how to select between the timing analyzers • Introduced the TimeQuest flow with PrimeTime	Introduction of the TimeQuest Timing Analyzer updated in this chapter.
May 2006 v6.0.0	Chapter title changed to *Synopsys PrimeTime Support*. Minor updates for the Quartus II software version 6.0.0.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
August 2005 v5.0.1	Minor text updates.	—
May 2005 v5.0.0	New functionality for Quartus II software 5.0.0	—
December 2004 v2.0	• Chapter 6 Synopsys PrimeTime moved to section III Volume 1. • New functionality for Quartus II software 4.2.	—

Section III. Power Estimation and Analysis

As FPGA designs grow larger and processes continue to shrink, power becomes an ever-increasing concern. When designing a printed circuit board, the power consumed by a device needs to be accurately estimated to develop an appropriate power budget, and to design the power supplies, voltage regulators, heat sink, and cooling system.

The Quartus® II software allows you to estimate the power consumed by your current design during timing simulation. The power consumption of your design can be calculated using the Microsoft Excel-based power calculator, or the Simulation-Based Power Estimation features in the Quartus II software. This section explains how to use both.

This section includes the following chapter:

- Chapter 10, PowerPlay Power Analysis



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

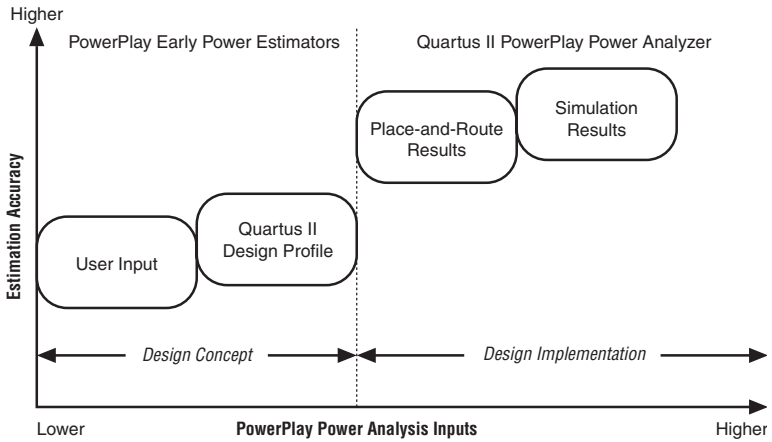
10. PowerPlay Power Analysis

QII53013-7.2.0

Introduction

As designs grow larger and process technology continues to shrink, power becomes an increasingly important design consideration. When designing a printed circuit board (PCB), the power consumed by a device needs to be accurately estimated to develop an appropriate power budget and to design the power supplies, voltage regulators, heat sink, and cooling system. The PowerPlay power analysis tools, made available by Altera®, provide improved power consumption accuracy and the ability to estimate power consumption from early design concept through design implementation, as shown in Figure 10–1.

Figure 10–1. PowerPlay Power Analysis



Depending where you are in your design cycle and the accuracy of the estimation required, you can either use the PowerPlay Early Power Estimator spreadsheet or the PowerPlay Power Analyzer Tool in the Quartus® II software. You can use the PowerPlay Early Power Estimator spreadsheet during the board design and layout phase to obtain a power estimate and then design for proper power management. The PowerPlay Power Analyzer Tool is used to obtain an accurate estimation of power after the design is complete, ensuring that thermal and supply budgets are not violated.

You can estimate power consumption for Arria™ GX, Stratix® series devices, Cyclone® series devices, HardCopy® II, and MAX® II devices with the Microsoft Excel-based PowerPlay Early Power Estimator spreadsheet or the PowerPlay Power Analyzer Tool.



For more information about acquiring the PowerPlay Power Estimator spreadsheet for Arria GX, Stratix series devices, Cyclone series, HardCopy II, and MAX II devices and its use, refer to www.altera.com/support/devices/estimator/pow-powerplay.html.

This chapter discusses the following topics:

- "Quartus II Early Power Estimator File"
- "Types of Power Analyses" on page 10–6
- "Factors Affecting Power Consumption" on page 10–6
- "Using the PowerPlay Power Analyzer" on page 10–23

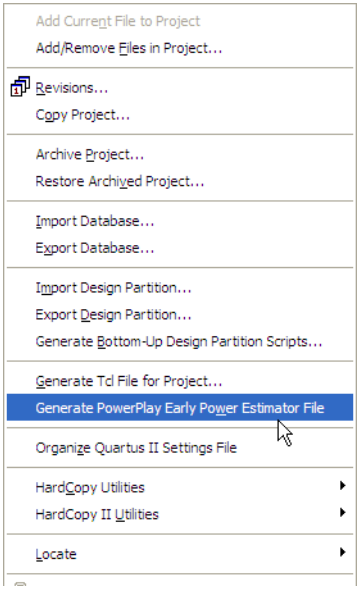
Quartus II Early Power Estimator File

When entering data into the Early Power Estimator spreadsheet, you must enter the device resources, operating frequency, toggle rates, and other parameters. This requires familiarity with the design. If you do not have an existing design, you must estimate the number of device resources used in your design and enter it manually.

If you already have an existing design or a partially completed design, the power estimator file that is generated by the Quartus II software can aid in completing the PowerPlay Early Power Estimator spreadsheet.

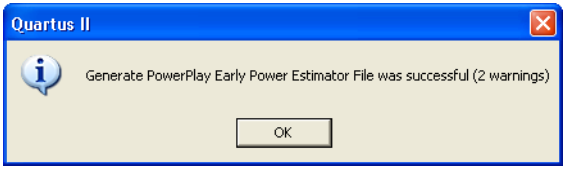
To generate the power estimation file, you must first compile your design in the Quartus II software. After compilation is complete, on the Project menu, click **Generate PowerPlay Early Power Estimator File** (Figure 10–2). This command instructs the Quartus II software to write out a power estimator Comma-Separated Value (**.csv**) file (or a text **[.txt]** file for older device families).

Figure 10–2. Generate PowerPlay Early Power Estimator File Option



After the Quartus II software successfully generates the power estimator file, a message appears (Figure 10–3).

Figure 10–3. Generate PowerPlay Early Power Estimator File Message



The power estimator file is named <*name of Quartus II project*> **_early_pwr.csv**. Figure 10–4 is an example of the contents of a power estimation file generated by the Quartus II software version 7.2 using a Stratix II device.

Figure 10–4. Example of Power Estimation File

EARLY_POWER_ESTIMATOR_FILE_FORMAT_VERSION	6								
QUARTUS_II_VERSION	7.2 Internal Build 105 07/05/2007 SJ Full Version								
PROJECT	Multaccum								
REVISION	Multaccum								
PROJECT_FILE	C:/Pipeline/Multiplier_Accum/Multaccum.qpf								
TIME	Mon Jul 09 14:28:12 2007								
TIME_SECONDS	1184016492								
FAMILY	Stratix II								
DEVICE	EP2S30								
PACKAGE	FBGA								
PART	EP2S30F672C3								
POWER_USE_DEVICE_CHARACTERISTICS	TYPICAL								
POWER_AUTO_COMPUTE_TJ	ON								
POWER_TJ_VALUE	25								
POWER_USE_CUSTOM_COOLING_SOLUTION	OFF								
POWER_PRESET_COOLING_SOLUTION	23 mm heat sink with 200 LFpM airflow								
POWER_BOARD_THERMAL_MODEL	None (Conservative)								
POWER_USE_TA_VALUE	25								
POWER_BOARD_TEMPERATURE	-1								
POWER_OJC_VALUE	0.21								
POWER_OCS_VALUE	0.1								
POWER_OSA_VALUE	2.8								
POWER_OJB_VALUE	-1								
VCCIO	1	3.3	2	3.3	3	3.3	4	3.3	
RAIL_VOLTAGES	VCCINT	1.2	VCCPD	3.3					
BLOCK	I/O	count	34	avg_toggle	21588235	avg_toggle	0.423299	avg_fanout	
BLOCK	I/O	count	5	avg_toggle	99600000	avg_toggle	1.952941	avg_fanout	
BLOCK	I/O	count	1	avg_toggle	21000000	avg_toggle	0.411765	avg_fanout	
BLOCK	I/O	count	51	avg_toggle	23117647	avg_toggle	0.453287	avg_fanout	
BLOCK	I/O	count	21	avg_toggle	98000000	avg_toggle	1.921569	avg_fanout	
BLOCK	DSP block output	count	16	avg_toggle	23893557	avg_toggle	0.468501	avg_fanout	
BLOCK	Combinational cell	count	6892	avg_toggle	66444042	avg_toggle	1.302824	avg_fanout	1.0293
BLOCK	Register cell	count	2018	avg_toggle	24156448	avg_toggle	0.473656	avg_fanout	6.6204
BLOCK	Clock control block	count	1	avg_toggle	1.02E+08	avg_toggle	2	avg_fanout	20



The power estimator file is named <*name of Quartus II project*> **_early_pwr.txt** for older device families.

The PowerPlay Early Power Estimator spreadsheet includes the Import Data macro that parses the information in the power estimation file and transfers it into the spreadsheet. If you do not want to use the macro, you can transfer the data into the Early Power Estimator spreadsheet manually.

If the existing Quartus II project represents only a portion of your full design, you should enter the additional resources used in the final design manually. Therefore, you can edit the spreadsheet and add additional device resources after importing the power estimation file information.

PowerPlay Early Power Estimator File Generator Compilation Report

After successfully generating the power estimation file, a PowerPlay Early Power Estimator File Generator report is created under the Compilation Report section. This report is divided into the different sections, such as Summary, Settings, Generated Files, Confidence Metric Details, and Signal Activities.

For more information about the PowerPlay Early Power Estimator File Generator report, refer to "PowerPlay Power Analyzer Compilation Report" on page 10–39.

Table 10–1 lists the main differences between the PowerPlay Early Power Estimator and the PowerPlay Power Analyzer.

Table 10–1. Comparison of PowerPlay Early Power Estimator and PowerPlay Power Analyzer

Characteristic	PowerPlay Early Power Estimator	PowerPlay Power Analyzer
Phase in the design cycle	Any time	After fitting
Tool requirements	Spreadsheet program/Quartus II software	Quartus II software
Accuracy	Medium	Medium to very high
Data inputs	• Resource usage estimates • Clock requirements • Environmental conditions • Toggle Rate	• Design after fitting • Clock requirements • Register transfer level (RTL) simulation results (optional) • Post-fitting simulation results (optional) • Signal activities per node or entity (optional) • Signal activity defaults • Environmental conditions
Data outputs *(1)*	• Total thermal power dissipation • Thermal static power • Thermal dynamic power • Off-chip power dissipation • Voltage supply currents *(2)*	• Total thermal power • Thermal static power • Thermal dynamic power • Thermal I/O power • Thermal power by design hierarchy • Thermal power by block type • Thermal power dissipation by clock domain • Off-chip (non-thermal) power dissipation • Voltage supply currents *(2)*

Notes to Table 10–1:

(1) Early Power Estimator output varies by device family as some features may not be available.

(2) Available only for Arria GX, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families.

The results of the Power Analyzer are only an estimation of power, not a specification. The purpose of the estimation is to help establish a guide for the design's power budget. Altera recommends that the actual power be measured on the board. You must measure the device's total dynamic current during device operation, because the estimate is very design dependent and depends on many variable factors, including input vector quantity, quality, and exact loading conditions of a PCB design. Static power consumption must not be based on empirical observation. The values reported by the Power Analyzer or datasheet must be used because the devices tested may not exhibit worst-case behavior.

Types of Power Analyses

Understanding the uses of power analysis and the factors affecting power consumption help you use the Power Analyzer effectively. Power analysis meets two significant planning requirements:

- **Thermal planning**: The designer must ensure that the cooling solution is sufficient to dissipate the heat generated by the device. In particular, the computed junction temperature must fall within normal device specifications.
- **Power supply planning**: Power supplies must provide adequate current to support device operation.

The two types of analyses are closely related because much of the power supplied to the device is dissipated as heat from the device. However, in some situations, the two types of analyses are not identical. For example, when you use terminated I/O standards, some of the power drawn from the FPGA device power supply is dissipated in termination resistors, rather than in the FPGA.

Power analysis also addresses the activity of the design over time as a factor that impacts the power consumption of the device. Static power is defined as the power consumed regardless of design activity. Dynamic power is the additional power consumed due to signal activity or toggling.

Factors Affecting Power Consumption

This section describes the factors affecting power consumption. Understanding these factors lets you use the Power Analyzer and interpret its results effectively.

Device Selection

Different device families have different power characteristics. Many parameters affect the device family power consumption, including choice of process technology, supply voltage, electrical design, and device

architecture. For example, the Cyclone II device family architecture was designed to consume less static power than the high-performance, full-featured, Stratix II device family.

Power consumption also varies within a single device family. A larger device typically consumes more static power than a smaller device in the same family, due to its larger transistor count. Dynamic power can also increase with device size in devices that employ global routing architectures, such as the MAX device family. Stratix, Cyclone, and MAX II devices do not exhibit significantly increased dynamic power as device size increases.

The choice of device package also affects the device's ability to dissipate heat. This can impact your cooling solution choice required to meet junction temperature constraints.

Finally, process variation can affect power consumption. Process variation primarily impacts static power, since sub-threshold leakage current varies exponentially with changes in transistor threshold voltage. As a result, it is critical to consult device specifications for static power and not rely on empirical observation. Process variation weakly affects dynamic power.

Environmental Conditions

Operating temperature primarily affects device static power consumption. Higher junction temperatures result in higher static power consumption. The device thermal power and cooling solution that you use must result in the device junction temperature remaining within the maximum operating range for that device.

The main environmental parameters affecting junction temperature are the cooling solution and ambient temperature.

Air Flow

Air flow is a measure of how quickly heated air is removed from the vicinity of the device and replaced by air at ambient temperature. This can either be specified as "still air" when no fan is used, or as the linear feet per minute rating of the fan used in the system. Higher air flow decreases thermal resistance.

Heat Sink and Thermal Compound

A heat sink allows more efficient heat transfer from the device to the surrounding area because of its large surface area exposed to the air. The thermal compound that interfaces the heat sink to the device also

influences the rate of heat dissipation. The case-to-ambient thermal resistance (θ_{CA}) parameter describes the cooling capacity of the heat sink and thermal compound employed at a given airflow. Larger heat sinks and more effective thermal compounds reduce θ_{CA}.

Ambient Temperature

The junction temperature of a device is equal to:

$$T_{Junction} = T_{Ambient} + P_{Thermal} \cdot \theta_{JA}$$

where θ_{JA} is the total thermal resistance from the device transistors to the environment, having units of degrees Celsius per Watt. The value θ_{JA} is equal to the sum of the junction-to-case (package) thermal resistance (θ_{JC}) and the case-to-ambient thermal resistance (θ_{CA}) of your cooling solution.

Board Thermal Model

The thermal resistance of the path through the board is referred to as the junction-to-board thermal resistance (θ_{JB}) (the units are in degrees Celsius per Watt). This is used in conjunction with the board temperature, as well as the top-of-chip θ_{JA} and ambient temperatures, to compute junction temperature.

Design Resources

The design resource used greatly affects power consumption.

Number, Type, and Loading of I/O Pins

Output pins drive off-chip components, resulting in high-load capacitance that leads to a high-dynamic power per transition. Terminated I/O standards require external resistors that generally draw constant (static) power from the output pin.

Number and Type of Logic Elements, Multiplier Elements, and RAM Blocks

A design with more logic elements (LEs), multiplier elements, and memory blocks tends to consume more power than a design with fewer such circuit elements. Also, the operating mode of each circuit element affects its power consumption. For example, a digital signal processing (DSP) block performing 18×18 multiplications and a DSP block performing multiply-accumulate operations consume different amounts

of dynamic power due to different amounts of internal capacitance being charged on each transition. Static power is also affected, to a small degree, by the operating mode of a circuit element.

Number and Type of Global Signals

Global signal networks span large portions of the device and have high capacitance, resulting in significant dynamic power consumption. The type of global signal is important as well. For example, Stratix II devices support several kinds of global clock networks that span either the entire device or a specific portion of the device (a regional clock network covers a quarter of the device). Clock networks that span smaller regions have lower capacitance and therefore, tend to consume less power. In addition, the location of the logic array blocks (LABs) that are driven by the clock network can have an impact, because the Quartus II software automatically disables unused branches of a clock.

Signal Activities

The final important factor in estimating power consumption is the behavior of each signal in the design. The two vital statistics are the toggle rate and the static probability.

The toggle rate of a signal is the average number of times that the signal changes value per unit time. The units for toggle rate are transitions per second, and a transition is a change from `1` to `0` or `0` to `1`.

The static probability of a signal is the fraction of time that the signal is logic `1` during the period of device operation that is being analyzed. Static probability ranges from `0` (always at ground) to `1` (always at logic high).

Dynamic power increases linearly with the toggle rate as the capacitive load is charged more frequently for logic and routing. The Quartus II models assume full rail-to-rail switching. For high toggle rates, especially on circuit output I/O pins, the circuit can transition before fully charging downstream capacitance. The result is a slightly conservative prediction of power by the Quartus II PowerPlay Power Analyzer.

The static power consumed by both routing and logic can sometimes be affected by the static probabilities of their input signals. This effect is due to state-dependent leakage, and has a larger affect on smaller process geometries. The Quartus II software models this effect on devices at 90 nm (or smaller) if it is deemed important to the power estimate. The static power also varies with the static probability of a logic `1` or `0` on the I/O pin when output I/O standards drive termination resistors.

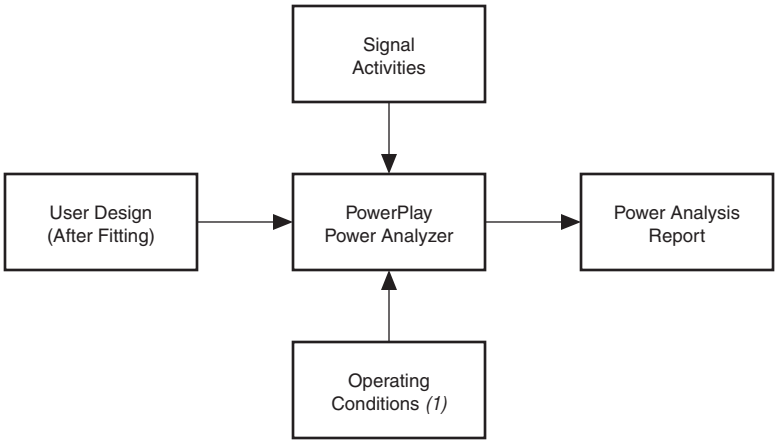


To get accurate results from power analysis, the signal activities that are used for analysis must be representative of the actual operating behavior of the design. Inaccurate signal toggle rate data is the largest source of power estimation error.

PowerPlay Power Analyzer Flow

The PowerPlay Power Analyzer supports accurate and representative power estimation by letting you specify all the important design factors affecting power consumption. Figure 10–5 shows the high-level Power Analyzer flow.

Figure 10–5. PowerPlay Power Analyzer High-Level Flow



Note to Figure 10–5:

(1) Operating condition specifications are available only for the Arria GX devices, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families.

The PowerPlay Power Analyzer requires that your design is synthesized and fit to the target device. Therefore, the Power Analyzer knows both the target device and how the design is placed and routed on the device. The electrical standard used by each I/O cell and the capacitive load on each I/O standard must be specified in the design to obtain accurate I/O power estimates.

Operating Conditions

For the Arria GX, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families, you can specify the operating conditions for power analysis in the Quartus II software.

The following settings are available in the **Settings** dialog box:

- **Device power characteristics**—Should the Power Analyzer assume typical silicon or maximum power silicon? The typical setting is useful for comparing to empirical data measured on an average unit. Worst-case data provides a boundary to the worst-case device that you could receive.
- **Selectable Core Voltage**—You can select a suitable core supply voltage for your design based on performance and power requirements using the **Core Supply Voltage** option, available for the latest devices with variable voltage support. The power consumption of a device is heavily dependent on the voltage, so it is very important to choose the right core supply voltage for your design. The core supply voltage provides power to device logic resources such as logic array blocks (LABs), MLABs, DSP functions, memory, and interconnects.
- **Environmental conditions and junction temperature**—By default, the Power Analyzer automatically computes the junction temperature based on the specified ambient temperature and the cooling solution that you selected from a list. For a more accurate analysis, enter the thermal resistance of your cooling solution. For some cooling solutions, such as a heat sink with no forced airflow, the thermal resistance varies with the amount of thermal power that is dissipated. Air convection increases as the difference between the device temperature and the ambient temperature increases, reducing thermal resistance. When entering a thermal resistance in such cases, it is important to use the thermal resistance that occurs when the heat flow (Q) is equal to the thermal power generated by the device. You can also specify a junction temperature in the PowerPlay Power Analyzer. However, Altera does not recommend this because the PowerPlay Power Analyzer provides more accurate results by computing the junction temperature.

- **Board Thermal Modeling**—If you want the Power Analyzer thermal model to take the θ_{JB} into consideration, set the board thermal model to either **Typical** or **Custom**. This feature produces more accurate thermal power estimation.

  A **Typical** board thermal model automatically sets θ_{JB} to a value based on the package and device selected. You only need to specify a board temperature. If you choose a **Custom** board thermal model,

you must specify a value for θ_{JB} and a board temperature. If you do not want the PowerPlay Power Analyzer thermal model to take the θ_{JB} resistance into consideration, set the **Board thermal model** option to **None** (conservative). In this case, the path through the board and power dissipation is not considered, and a more conservative thermal power estimate is obtained.

The **Board thermal model** option is only available if you select the **Auto compute junction temperature** option with the pre-set cooling solution set to some heat sink solution option or custom solution. This option is disabled when a cooling solution with no heat sink is selected, as thermal conduction through the board is included in the θ_{JA} value used to compute a junction temperature in that case.

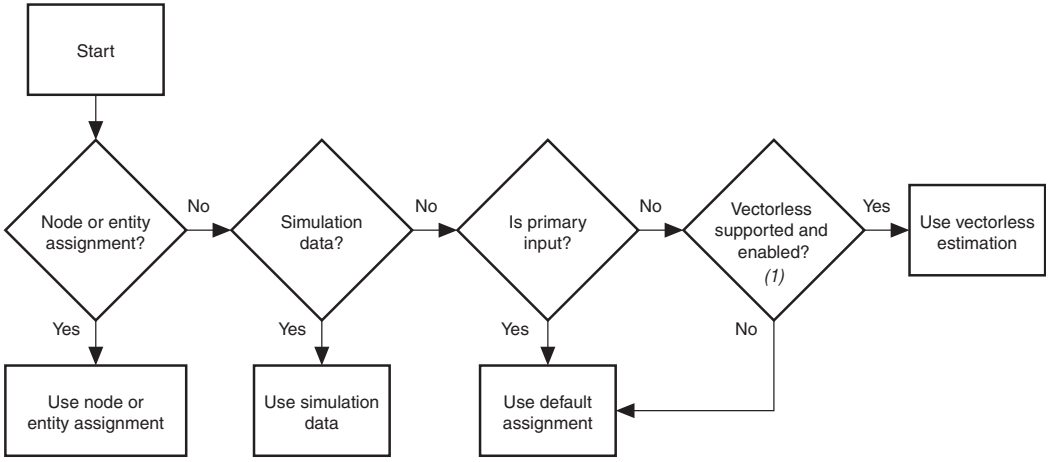
Signal Activities Data Sources

The Power Analyzer provides a flexible framework for specifying signal activities. This reflects the importance of using representative signal activity data during power analysis. You can use the following sources to provide information about signal activity:

- Simulation results
- User-entered node, entity, and clock assignments
- User-entered default toggle rate assignment
- Vectorless estimation

The PowerPlay Power Analyzer lets you mix and match the signal activity data sources on a signal-by-signal basis. Figure 10–6 shows the priority scheme. The data sources are described in the following sections.

Figure 10–6. Signal Activity Data Source Priority Scheme



Note to Figure 10–6:

(1) Vectorless estimation is available only for the Arria GX, Stratix III, Stratix II, Stratix II GX, Cyclone II, HardCopy II, and MAX II device families.

Simulation Results

The Power Analyzer directly reads the waveforms generated by a design simulation. The static probability and toggle rate for each signal is calculated from the simulation waveform. Power analysis is most accurate when simulations are generated using representative input stimuli.

The Power Analyzer reads the results generated by the following simulators:

- Quartus II Simulator
- ModelSim® VHDL, Active HDL, ModelSim Verilog HDL, ModelSim-Altera VHDL, ModelSim-Altera Verilog
- NC-Verilog, NC-VHDL
- VCS

Signal activity and static probability information are stored in a Signal Activity File (**.saf**) or may be derived from a Value Change Dump File (**.vcd**), described in "Signal Activities" on page 10–9. The Quartus II simulator generates a Signal Activity File (SAF) or a Value Change Dump (VCD) file which is then read by the Power Analyzer.

For third-party simulators, use the Quartus II EDA Tool Settings for Simulation to specify a **Generate Value Change Dump** file script. These scripts instruct the third-party simulators to generate a VCD file that encodes the simulated waveforms. The Quartus II Power Analyzer reads this file directly to derive toggle rate and static probability data for each signal.

Third-party EDA simulators, other than those listed above, can generate a VCD file that can then be used with the Power Analyzer. For those simulators, it is necessary to manually create a simulation script to generate the appropriate Value Change Dump File.



You can use a SAF or VCD file created for power analysis to optimize the design for power during fitting by utilizing the appropriate settings in the **PowerPlay power optimization** list, available in **Fitter Settings** page of the **Settings** dialog box.

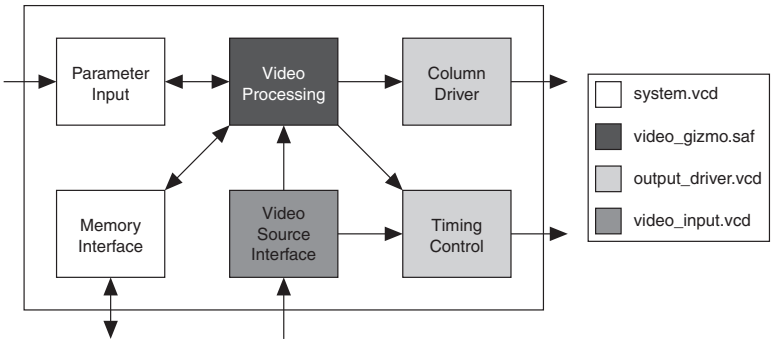


For more information about power optimization, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Using Simulation Files in Modular Design Flows

A common design practice is to create modular or hierarchical designs in which you develop each design entity separately and then instantiate it in a higher-level entity, forming a complete design. Simulation is performed on a complete design or on each modular design for verification. The Quartus II PowerPlay Power Analyzer Tool supports modular design flows when reading the signal activities generated from these simulation files, as shown in Figure 10–7.

Figure 10–7. Modular Simulation Flow

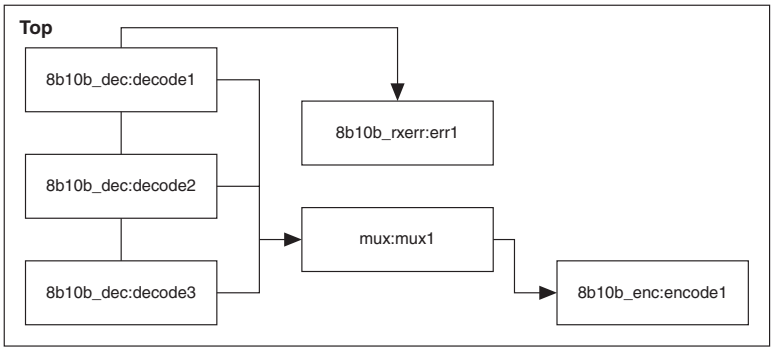


When specifying a simulation file, an associated design entity name may be given, such that the signal activities derived from the simulation file (VCD file or SAF) can be imported into the Power Analyzer for that particular design entity. The PowerPlay Power Analyzer Tool also supports the specification of multiple SAFs for power analysis with each having an associated design entity name to allow the integration of partial design simulations into a complete design power analysis. When specifying multiple SAFs for your design, it is possible that more than one simulation file will contain signal activity information for the same signal. In the case where multiple SAFs are applied to the same design entity, the signal activity used in the power analysis is the equal-weight arithmetic average of each SAF. Also in the case where multiple simulation files are applied to design entities at different levels in the design hierarchy, the signal activity used in the power analysis is derived from the simulation file that is applied to the most specific design entity.

Figure 10–8 shows an example of a hierarchical design. The design Top consists of three 8b/10b Decoders, followed by a multiplexer whose output is then encoded again before being output from the design. There is also an error-handling module that handles any 8b/10b decoding errors. The top-level module, called Top, automatically contains the design's top-level entity and any logic not defined as part of another module. The design file for the top-level module may be just a wrapper

for the hierarchical entities below it, or it may contain its own logic. The following usage scenarios show common ways that you may simulate your design and import SAFs into the PowerPlay Power Analyzer Tool.

Figure 10–8. Example Hierarchical Design



Complete Design Simulation

You can simulate the entire design Top, generating a VCD file if you use a third-party simulator, or generating a SAF or VCD if you use the Quartus II Simulator. The VCD file or SAF can then be imported (specifying Entity Top) into the power analyzer. The resulting power analysis uses all the signal activities information from the generated VCD file or SAF, including those that apply to submodules, such as `decode[1-3]`, `err1`, `mux1`, and `encode1`.

Modular Design Simulation

You can simulate submodules of the design Top independently, and then import all of the resulting SAFs into the Power Analyzer. For example, you may simulate the `8b10b_dec` independent of the entire design, as well as multiplexer, `8b10b_rxerr`, and `8b10b_enc`. You can then import the VCD file or SAF generated from each simulation by specifying

the appropriate instance name. For example, if the files produced by the simulations are **8b10b_dec.vcd**, **8b10b_enc.vcd**, **8b10b_rxerr.vcd**, and **mux.saf**, the import specifications in Table 10–2 are used.

Table 10–2. Import Specifications

File Name	Entity
8b10b_dec.vcd	`Top|8b10b_dec:decode1`
8b10b_dec.vcd	`Top|8b10b_dec:decode2`
8b10b_dec.vcd	`Top|8b10b_dec:decode3`
8b10b_rxerr.vcd	`Top|8b10b_rxerr:err1`
8b10b_enc.vcd	`Top|8b10b_enc:encode1`
mux.saf	`Top|mux:mux1`

The resulting power analysis applies the simulation vectors found in each file to the assigned entity. Simulation provides signal activities for the pins and for the outputs of functional blocks. If the inputs to an entity instance are input pins for the entire design, the simulation file associated with that instance does not provide signal activities for the inputs of that instance. For example, an input to an entity such as `mux1` has its signal activity specified at the output of one of the decode entities.

Multiple Simulations on the Same Entity

You can perform multiple simulations of an entire design or specific modules of a design. For example, in the process of verifying the Top design, you may have three different simulation testbenches: one for normal operation, and two for corner cases. Each of these simulations produces a separate VCD file or SAF. In this case, apply the different VCD file or SAF names to the same top-level entity, shown in Table 10–3.

Table 10–3. Multiple Simulation File Names and Entities

File Name	Entity
normal.saf	`Top`
corner1.vcd	`Top`
corner2.vcd	`Top`

The resulting power analysis uses an arithmetic average of the signal activities calculated from each simulation file to obtain the final signal activities used. Thus, if a signal `err_out` has a toggle rate of 0 toggles per

second in **normal.saf**, 50 toggles per second in **corner1.vcd**, and 70 toggles per second in **corner2.vcd**, the final toggle rate that is used in the power analysis is 40 toggles per second.

Overlapping Simulations

You can perform a simulation on the entire design Top and more exhaustive simulations on a submodule, such as `8b10b_rxerr`. Table 10–4 shows the import specification for overlapping simulations.

Table 10–4. Overlapping Simulation Import Specifications

File Name	Entity
full_design.vcd	`Top`
error_cases.vcd	`Top|8b10b_rxerr:err1`

In this case, signal activities from **error_cases.vcd** are used for all of the nodes in the generated SAF, and signal activities from **full_design.vcd** are used for only those nodes that do not overlap with nodes in **error_cases.vcd**. In general, the more specific hierarchy (the most bottom-level module) is used to derive signal activities for overlapping nodes.

Partial Simulations

You can perform a simulation where the entire simulation time is not applicable to signal activity calculation. For example, suppose you run a simulation for 10,000 clock cycles and you reset the chip for the first 2,000 clock cycles. If the signal activity calculation is performed over all 10,000 cycles, the toggle rates are typically only 80% of their steady state value (since the chip is in reset for the first 20% of the simulation). In this case, you should specify the useful parts of the VCD file for power analysis. The Limit VCD Period option enables you to specify a start and end time to be used when performing signal activity calculations.

Node Name Matching Considerations

Node name mismatches happen when you have SAFs or VCD files applied to entities other than the top-level entity. In a modular design flow, the gate-level simulation files created in different Quartus II software projects may not match their node names properly with the current Quartus II project.

For example, if you have a file named **8b10b_enc.vcd**, which was generated in a separate project called **8b10b_enc** and is simulating the `8b10b` encoder, and you import that VCD file into another project called **Top**, you may encounter name mismatches when applying the VCD file to the `8b10b_enc` module in the **Top** project. This is because all of the combinational nodes in the **8b10b_enc.vcd** file may be named differently in the **Top** project.

You can avoid name mismatching by using only register transfer level (RTL) simulation data, where register names usually do not change, or by using an incremental compile flow that preserves node names in conjunction with a gate-level simulation. To ensure the best accuracy, Altera recommends using an incremental compile flow to preserve your design's node names.



For more information about the incremental compile flow, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Glitch Filtering

The Power Analyzer defines a glitch as two signal transitions that are so closely spaced in time that the pulse, or glitch, occurs faster than the logic and routing circuitry can respond. The output of a transport delay model simulator (the default mode of the Quartus II simulator) generally contains glitches for some signals. The device's logic and routing structures form a low-pass filter that filters out glitches that are tens to hundreds of picoseconds long, depending on the device family.

Some third-party simulators use different simulator models than the transport delay model as default. Different models cause differences in signal activity estimation and power estimation. The inertial delay model, which is the ModelSim default model, filters out many more glitches than the transport delay model; therefore, it usually yields a lower power estimate. Altera recommends using the transport simulation model when using the Quartus II glitch filtering support with third-party simulators. If the inertial simulation model is used, simulation glitch filtering has little effect.



For more information about how to set the simulation model type for your specific simulator, refer to the Quartus II Help.

Glitch filtering in a simulator can also filter a glitch on one LE (or other circuit element) output from propagating to downstream circuit elements so that the glitch will not affect simulated results. This prevents a glitch on one signal from producing non-physical glitches on all downstream logic, which would result in a signal toggle rate that is too high and a

power estimate that is too high. Circuit elements in which every input transition produces an output transition, including multipliers and logic cells configured to implement XOR functions, are especially prone to glitches. Therefore, circuits with many such functions can have power estimates that are too high when glitch filtering is not used.

Altera recommends that the glitch filtering feature be used to obtain the most accurate power estimates. For VCD files, the Power Analyzer flows support two types of glitch filtering, both of which are recommended for power estimation. In the first, glitches are filtered during simulation. To enable this level of glitch filtering in the Quartus II software for supported third-party simulators, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears.
3. Select the **Tool Name** to use for the simulation.
4. Turn on the **Enable glitch filtering** option.

To enable this level of glitch filtering in the Quartus II software using the Quartus II Simulator, refer to "Generating a SAF or VCD File Using the Quartus II Simulator" on page 10–24.

The second level of glitch filtering occurs while the Power Analyzer is reading the VCD file generated by the third-party simulator or Quartus II Simulator. Enable this level of glitch filtering by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **PowerPlay Power Analyzer Settings**. The **PowerPlay Power Analyzer Settings** page appears.
3. Under **Input File(s)**, turn on the **Perform glitch filtering on VCD files** option.

Altera recommends that you use both forms of glitch filtering.

The VCD file reader performs complementary filtering to the filtering performed during simulation and is often not as effective. While the VCD file reader can remove glitches on logic blocks, it has no way of determining how downstream logic and routing are affected by a given

glitch, and may not eliminate the impact of the glitch completely. Filtering the glitches during simulation avoids switching downstream routing and logic automatically.



When running simulation for design verification (rather than to produce input to the Quartus PowerPlay Power Analyzer), Altera recommends leaving glitch filtering turned off. This produces the most rigorous and conservative simulation from a functionality viewpoint. When performing simulation to produce input for the Quartus II PowerPlay Power Analyzer, Altera recommends turning on glitch filtering to produce the most accurate power estimates.

Node and Entity Assignments

You can assign specific toggle rates and static probabilities to individual nodes and entities in the design. These assignments have the highest priority, overriding data from all other signal activity sources.

Use the Assignment Editor or tool command language (Tcl) commands to make the **Power Toggle Rate** and **Power Static Probability** assignments. You can specify the power toggle rate as an absolute toggle rate in transitions using the **Power Toggle Rate** assignment or you can use the **Power Toggle Rate Percentage** assignment to specify a toggle rate relative to the clock domain of the assigned node for more specific assignment made in terms of hierarchy level.



If the **Power Toggle Rate Percentage** assignment is used, and the given node does not have a clock domain, a warning is issued and the assignment is ignored.



For more information about how to use the Assignment Editor in the Quartus II software, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

This method is appropriate for special-case signals where you have specific knowledge of the signal or entity being analyzed. For example, if you know that a 100-MHz data bus or memory output produces data that is essentially random (uncorrelated in time), you can directly enter a 0.5 static probability and a toggle rate of 50 million transitions per second.

Bidirectional I/O pins are treated specially. The combinational input port and the output pad for a given pin share the same name. However, those ports might not share the same signal activities. For the purpose of reading signal activity assignments, the Power Analyzer creates a distinct name `<node_name~output>` when the bidirectional signal is configured as an output and `<node_name~result>` when the signal is

configured as an input. For example, if a design has a bidirectional pin named MYPIN, assignments for the combinational input use the name MYPIN~result, and the assignments for the output pad use the name MYPIN~output.



When making the logic assignment in the Assignment Editor, you will not find the MYPIN~result and MYPIN~output node names in the Node Finder. Therefore, to make the logic assignment, you must manually enter the two differentiating node names to make the specific assignment for the input and output port of the bidirectional pin.

Timing Assignments to Clock Nodes

For clock nodes, the Power Analyzer uses the timing requirements to derive the toggle rate when neither simulation data nor user entered signal activity data is available.



f_{MAX} requirements specify full cycles per second, but each cycle represents a rising transition and a falling transition. For example, a clock f_{MAX} requirement of 100 MHz corresponds to 200 million transitions per second.

Default Toggle Rate Assignment

You can specify a default toggle rate for primary inputs and all other nodes in the design. The default toggle rate is used when no other method has specified the signal activity data.

The toggle rate can be specified in absolute terms (transitions per second) or as a fraction of the clock rate in effect for each particular node. The toggle rate for a given clock is derived from the timing settings for the clock. For example, if a clock is specified with an f_{MAX} constraint of 100 MHz and a default relative toggle rate of 20%, nodes in this clock domain transition in 20% of the clock periods, or 20 million transitions occur per second. In some cases, the Power Analyzer cannot determine the clock domain for a given node because there is either no clock domain for the node or it is ambiguous. In these cases, the Power Analyzer substitutes and reports a toggle rate of zero.

Vectorless Estimation

For some device families, the Power Analyzer automatically derives estimates for signal activity on nodes with no simulation or user-entered signal-activity data. Vectorless estimation is available and enabled by default for Arria GX, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families. Vectorless estimation statistically estimates the signal activity of a node based on the signal activities of all nodes feeding that node, and on the actual logic function that is implemented by the node. The PowerPlay Power Analyzer **Settings** dialog box lets you disable vectorless estimation. When enabled, vectorless estimation takes priority over default toggle rates. Vectorless estimation does not override clock assignments.



Vectorless estimation cannot derive signal activities for primary inputs. Vectorless estimation is generally accurate for combinational nodes, but not for registered nodes. Therefore, simulation data for at least the registered nodes and I/O nodes is needed for accuracy.

Using the PowerPlay Power Analyzer

For all flows that use the PowerPlay Power Analyzer, synthesize your design first and then fit it to the target device. You must either provide timing assignments for all clocks in the design or use a simulation-based flow to generate activity data. The I/O standard used on each device input or output and the capacitive load on each output must be specified in the design.

Common Analysis Flows

You can use the analysis flows in this section with the PowerPlay Power Analyzer. However, vectorless activity estimation is only available for some device families.

Signal Activities from Full Post-Fit Netlist (Timing) Simulation

This flow provides the highest accuracy because all node activities reflect actual design behavior, provided that supplied input vectors are representative of typical design operation. Results are better if the simulation filtered glitches. The disadvantage with this method is that simulation times can be long.

Signal Activities from RTL (Functional) Simulation, Supplemented by Vectorless Estimation

In this flow, simulation provides toggle rates and static probabilities for all pins and registers in the design. Vectorless estimation fills in the values for all the combinational nodes between pins and registers. This method yields good results, since vectorless estimation is accurate, given that the proper pin and register data is provided. This flow usually provides a compilation time benefit to the user in the third-party RTL Simulator.



RTL simulation may not provide signal activities for all registers in the post-fitting netlist because some register names may be lost during synthesis. For example, synthesis may automatically transform state machines and counters, thus changing the names of registers in those structures.

Signal Activities from Vectorless Estimation, User-Supplied Input Pin Activities

This option provides a low level of accuracy, because vectorless estimation for registers is not entirely accurate.

Signal Activities from User Defaults Only

This option provides the lowest degree of accuracy.

Generating a SAF or VCD File Using the Quartus II Simulator

While performing a timing or functional simulation using the Quartus II Simulator, you can generate a SAF or VCD file. These files store the toggle rate and static probability for each connected output signal based on the simulation vectors that are entered in the Vector Waveform File (**.vwf**) or the Vector File (**.vec**). You can use the SAF(s) or VCD file(s) as input to the PowerPlay Power Analyzer to estimate power for your design.

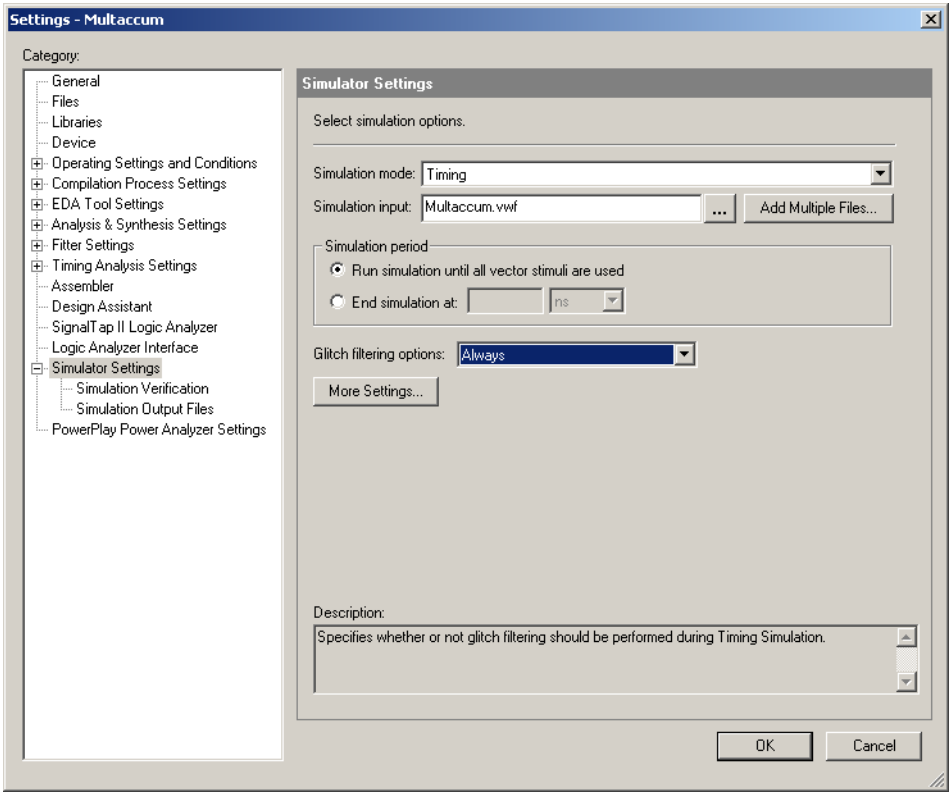


For more accurate results, Altera recommends that you use the SAF created from the Quartus II simulator as the input to the PowerPlay Power Analyzer.

To create a SAF or VCD file for your design, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulator Settings**. The **Simulator Settings** page appears (Figure 10–9).

Figure 10–9. Simulator Settings Page



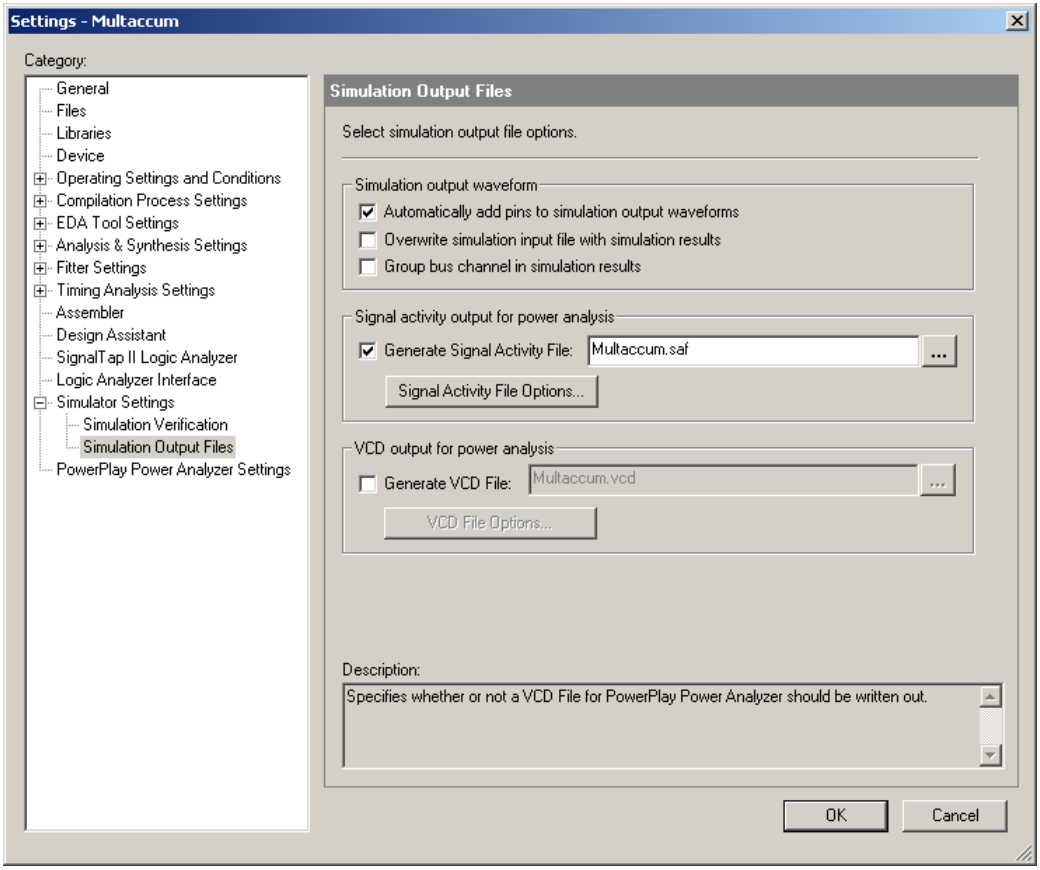
3. In the **Simulation mode** list, select either **Timing** or **Functional**. Refer to "Common Analysis Flows" on page 10–23 for a description of the difference in accuracy between the two types of simulation modes.

4. (Optional) Click **More Settings**. The **More Simulator Settings** dialog box appears.

5. (Optional) Turn on glitch filtering. To turn on glitch filtering, in the **Glitch filtering options** list, select **Always**.

6. In the **Category** list, click the ⊞ icon to expand **Simulator Settings** and select **Simulation Output Files** (Figure 10–10).

Figure 10–10. Simulator Output Files Page of the Settings Dialog Box



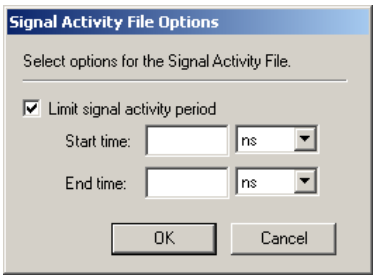
7. Turn on **Generate Signal Activity File** and enter the file name for the SAF file.

For more information about the Quartus II Simulator and how to create a SAF file, refer to the *Quartus II Simulator* chapter in volume 3 of the *Quartus II Handbook*.

When generating a VCD file from the Quartus Simulator, you must make sure that you add **all nodes** to the input vector wave file. Only the nodes that have been added to your vector file will be output to the Quartus-generated VCD file. This is not the case when generating a SAF. The Quartus II Simulator will create a SAF including all the internal nodes of your design even if the stimuli file contains only the input vectors for your simulation.

8. (Optional) Click **Signal Activity File Options**. The **Signal Activity File Options** dialog box appears (Figure 10–11).

Figure 10–11. Signal Activity File Options Dialog Box



9. (Optional) Turn on the **Limit signal activity period** option to specify the simulation period to use when calculating the signal activities.

   Power estimation can be performed for the entire simulation time or for a portion of the simulation time. This allows you to look at the power consumption at different points in your overall simulation without having to rework your testbenches. This feature is also useful when multiple clock cycles are necessary to initialize the state of the design, but you want to measure the signal activity only during the normal operation of the design, not during its initialization phase. You can specify the start time and end time in the **Signal Activity File Options** dialog box by turning on the **Limit signal activity period** option. Simulation information is used during this time interval only to calculate toggle rates and static probabilities. If no time interval is specified, the whole simulation is used to compute signal activity data.

10. After the simulation is complete, a SAF is generated with the specified filename and stored in the main project directory.

For more information about how to perform simulations in the Quartus II software, see the Quartus II Help.

Generating a VCD File Using a Third-Party Simulator

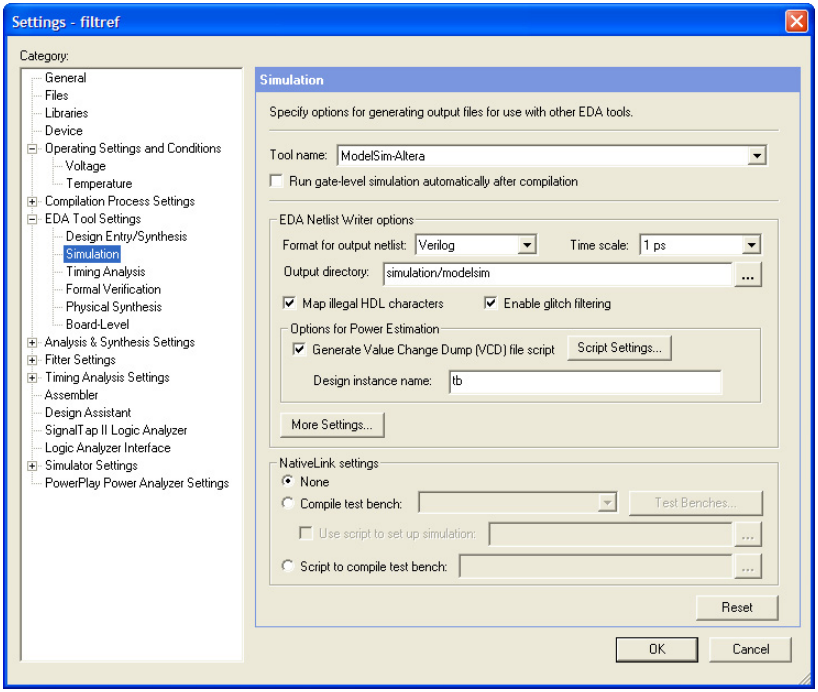
You can use other EDA simulation tools, such as the Model Technology™ ModelSim® software, to perform a simulation and create a VCD file. You can use this file as input to the PowerPlay Power Analyzer to estimate power for your design. To do this, you must tell the Quartus II software to generate a script file that is used as input to the third-party simulator. This script tells the third-party simulator to generate a VCD file that contains all the output signals. For more information about the supported third-party simulators, refer to "Simulation Results" on page 10–13.

To create a VCD file for your design, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.

2. In the **Category** list, select **Simulation**. The **Simulation** page appears, as shown in Figure 10–12.

Figure 10–12. Simulation Page of the Settings Dialog Box



3. In the **Tool name** list, select the appropriate EDA simulation tool.

4. In the **Format for output netlist** list, select **VHDL** or **Verilog**.

5. Turn on **Generate Value Change Dump (VCD) file script**.

   This turns on the **Map illegal HDL character** and **Enable glitch filtering** options.

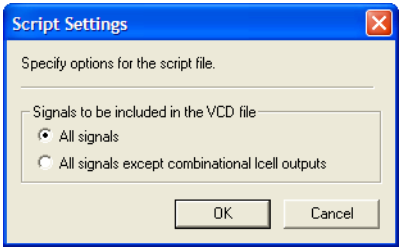
6. (Optional) **Map illegal HDL characters** ensures that all signals have legal names and that signal toggle rates are available later in the PowerPlay Power Analyzer.

7. (Optional) By turning on **Enable glitch filtering**, glitch filtering logic is the output when you generate an EDA netlist for simulation. This option is always available, regardless of whether or not you want to generate the VCD file scripts. For more information about glitch filtering, refer to "Glitch Filtering" on page 10–19.

When performing simulation using ModelSim, the **+nospecify** option given to the `vsim` command disables **specify** path delays and timing checks in ModelSim. By enabling glitch filtering on the **Simulation** page, the simulation models include **specify** path delays. Thus, ModelSim can fail to simulate a design if glitch filtering is enabled and the **+nospecify** option is specified. Altera recommends the removal of the **+nospecify** option from the ModelSim `vsim` command to ensure accurate simulation for power estimation.

8. Click **Script Settings**. The **Script Settings** dialog box appears, shown in Figure 10–13.

Figure 10–13. Script Settings Dialog Box



Select which signals should be output to the VCD file. With **All signals** selected, the generated script instructs the third-party simulator to write all connected output signals to the VCD file. With **All signals except combinational lcell outputs** selected, the

generated script tells the third-party simulator to write all connected output signals to the VCD file, except logic cell combinational outputs. You may not want to write all output signals to the file because the file can become extremely large (since its size depends on the number of output signals being monitored and the number of transitions that occur).

9. Click **OK**.

10. Type a name for your testbench in the **Design instance name** box.

11. Compile your design with the Quartus II software and generate the necessary EDA netlist and script that tells the third-party simulator to generate a VCD file.



For more information about NativeLink use, refer to *Section I. Simulation* in volume 3 of the *Quartus II Handbook*.

12. Perform a simulation with the third-party EDA simulation tool. Call the generated script in the simulation tool before running the simulation. The simulation tool generates the VCD file and places it in the project directory.

The following example provides step-by-step instructions to successfully produce a VCD file with the ModelSim software:

1. In the Quartus II software, on the Assignments menu, click **Settings**.

2. In the **Settings** dialog box, on the **Simulator Settings** page, choose the appropriate ModelSim selection in the **Tool Name** list, and turn on the **Generate Value Change Dump File Script** option.

3. To generate the VCD file, perform a full compilation.

4. In the ModelSim software, compile the files necessary for simulation.

5. Load your design by clicking **Start Simulation** on the Tools menu, or use the **vsim** command.

6. Source the Quartus II VCD script created in step 3 using the following command:
```
source <design>_dump_all_vcd_nodes.tcl
```

7. Run the simulation (for example, **run 2000ns** or r**un -all**).

8. Quit the simulation using the **quit -sim** command, if needed.

9. Exit the ModelSim software. If you do not exit the software, the ModelSim software may end the writing process of the VCD files improperly, resulting in a corrupted VCD file.

For more information about how to call the VCD file generation script in the respective third-party EDA simulation tools, refer to the Quartus II Help. For more information about how to perform simulations in other EDA simulation tools, see the relevant documentation for that tool.

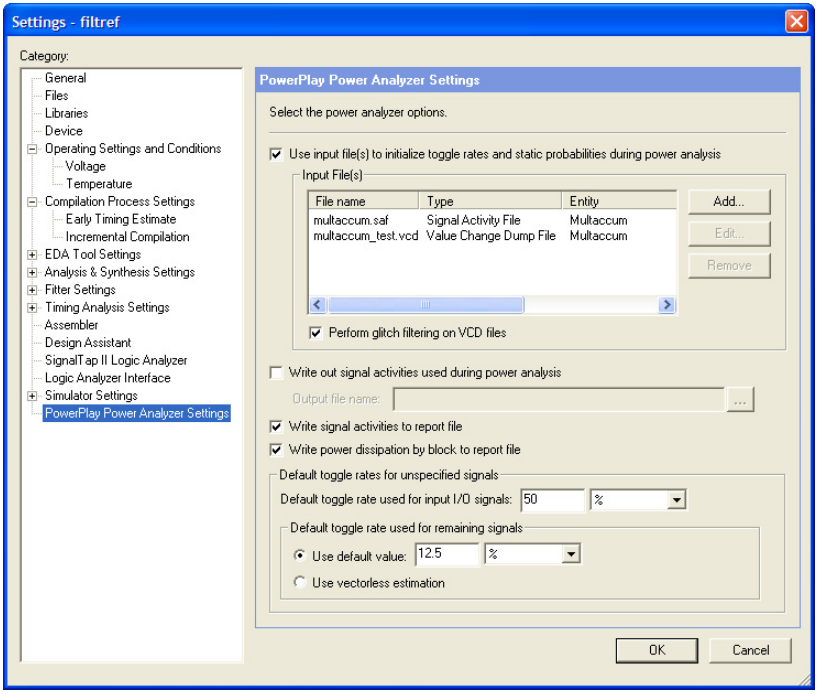
Running the PowerPlay Power Analyzer Using the Quartus II GUI

To run the PowerPlay Power Analyzer using the Quartus II GUI, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.

2. In the **Category** list, select **PowerPlay Power Analyzer Settings**, shown in Figure 10–14.

Figure 10–14. PowerPlay Power Analyzer Settings

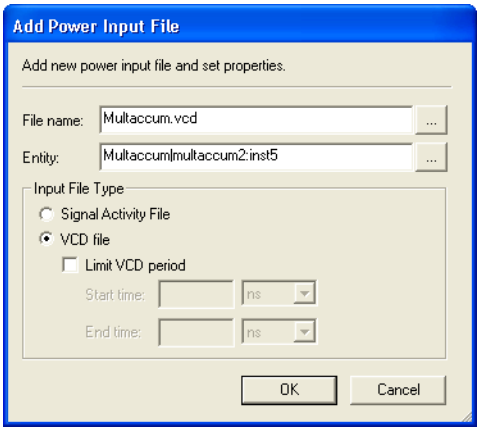


3. (Optional) If you want to use either SAF(s) or VCD file(s) or both as an input to the PowerPlay Power Analyzer, turn on **Use input file(s) to initialize toggle rates and static probabilities during power analysis**.

   (Optional) The **Edit** button allows you to change the settings for a selected file from the list. The **Remove** button allows you to remove a selected file from the list.

4. Click **Add**. The **Add Power Input File** dialog box appears, as shown in Figure 10–15.

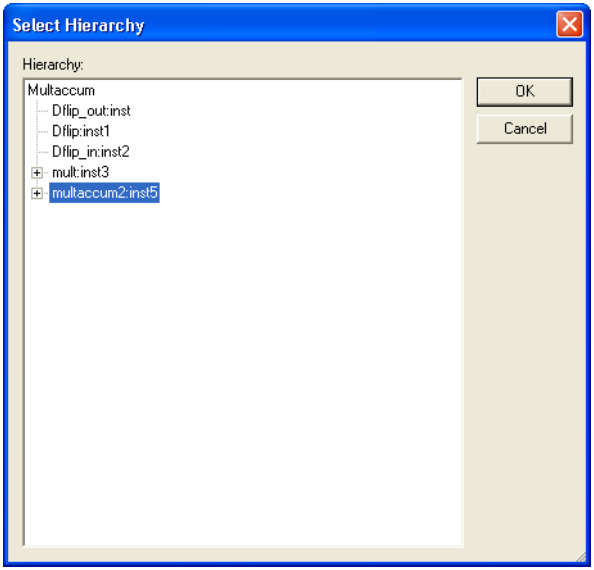
Figure 10–15. Add Power Input File Dialog Box



5. Add your SAF(s) or VCD file(s) by clicking the browse button for the **File name** box.

6. The **Entity** box enables you to specify the design entity (hierarchy) to which the entered power input file applies. To enter the entity, you can type in the box or browse through the list of your design entities. To browse your design entities, click the browse button. The **Select Hierarchy** dialog box appears, shown in Figure 10–16. You can specify multiple entities in the entity text box by using comma delimiters.

Figure 10–16. Select Hierarchy Dialog Box



7. You can specify whether the input file is a VCD file or SAF under **Input File Type**.

8. (Optional) **Limit VCD period** is enabled only when the **VCD file** is selected. This enables you to specify the simulation period to use when calculating the signal activities. For more information, refer to step 9 of "Generating a SAF or VCD File Using the Quartus II Simulator" on page 10–24.

9. Click **OK**.

10. Click **OK** in the **Add Power Input File** dialog box.

11. (Optional) Turn on **Perform glitch filtering on VCD files**. This option is recommended. For more information, refer to "Glitch Filtering" on page 10–19.

12. (Optional) Turn on **Write out signal activities used during power analysis**. In the **Output file name** list, select the output file name. This file contains all the signal activities information used during the power estimation of your design. This is recommended if you

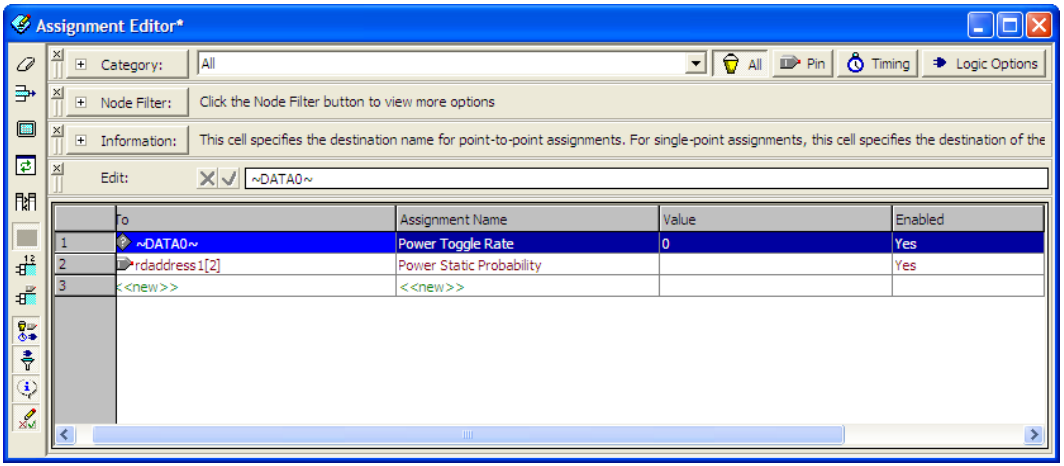
used a VCD file as input into the PowerPlay Power Analyzer, because it reduces the run time of any subsequent power estimation. You can use the generated SAF as input instead of the original VCD file.

13. (Optional) Turn on **Write signal activities to report file**.

14. (Optional) Turn on **Write power dissipation by block to report file** to enable the output of detailed thermal power dissipation by block to be included in the PowerPlay Power Analyzer report.

15. (Optional) You can also use the Assignment Editor to enter the Power Toggle Rate or Power Toggle Rate Percentage, and the Power Static Probability for a node or entity in your design, shown in Figure 10–17.

Figure 10–17. Assignment Editor *Notes (1), (2)*



Notes to Figure 10–17:
(1) The assignments made with the Assignment Editor override the values already existing in the SAF or VCD file.
(2) You can also use Tcl script commands to make these assignments.

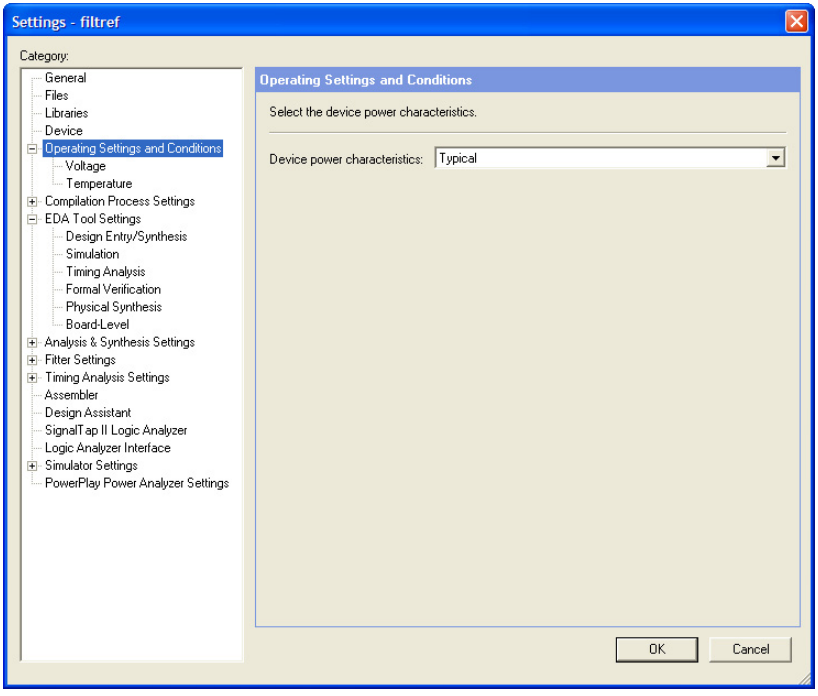
For more information about how to use the Assignment Editor in the Quartus II software, see the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*. For information about scripting, see the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

16. Specify the toggle rate in the **Default toggle rate used for input I/O signals** field. This toggle rate is used for all unspecified input I/O signal toggle rates regardless of whether or not the device family supports vectorless estimation. By default, its value is set to 12.5%. The default static probability for unspecified input I/O signals is 0.5 and cannot be changed.

17. Select either **Use default value** or **Use vectorless estimation** for Arria GX, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, or MAX II device families. For all other device families, only **Use default value** is available. This setting controls how the remainder of the unspecified signal activities are calculated. For more information, refer to "Vectorless Estimation" on page 10–23 and "Default Toggle Rate Assignment" on page 10–22.

18. In the **Category** list, select **Operating Settings and Conditions**. This option is available only for the Arria GX, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families (Figure 10–18).

Figure 10–18. Operating Conditions



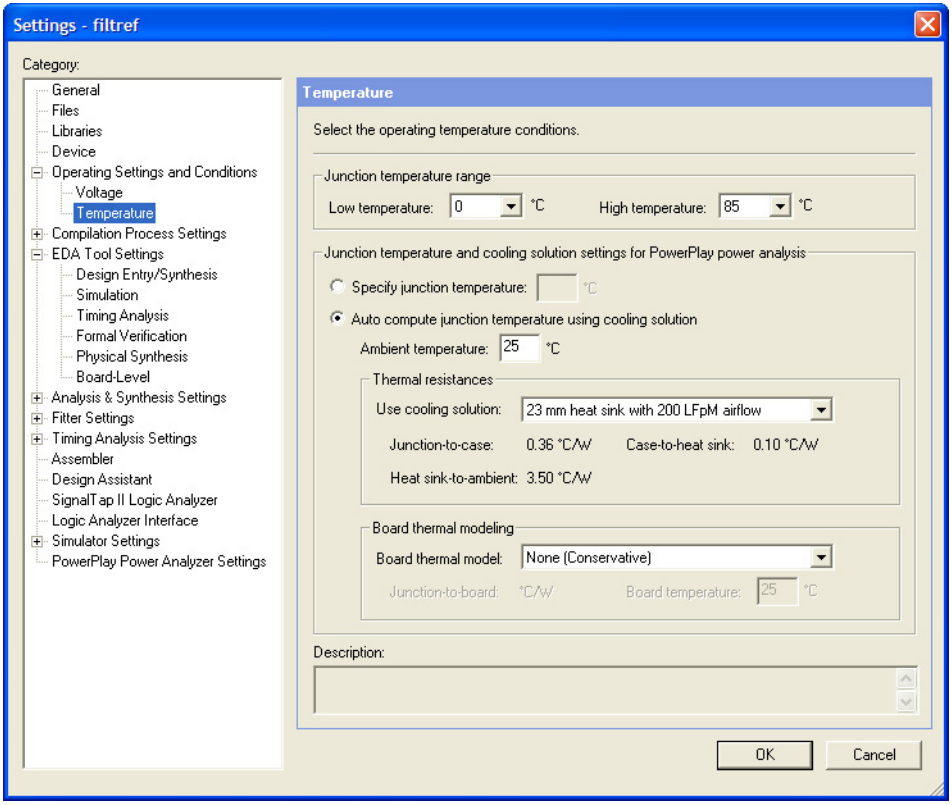
19. In the **Device power characteristics** list, select **Typical** or **Maximum**. The default is **Typical**.

20. In the **Category** list, click the ⊞ icon to expand **Operating Settings and Conditions** and click **Voltage**. The **Voltage** page appears.

21. For the devices with selectable core voltage support, in the **Core supply voltage** list, select the core supply voltage for your device. This option is available for the latest devices with variable voltage selection.

22. In the **Category** list, under **Operating Settings and Conditions,** select **Temperature**. The **Temperature** page appears (Figure 10–19).

Figure 10–19. Temperature Settings Page



23. Under **Junction temperature range**, specify a junction temperature in degrees Celsius and specify the junction temperature range. Select the **Low temperature** and **High temperature** range for your selected device.

24. Specify the junction temperature and cooling solution settings. You can select **Specify junction temperature** or **Auto compute junction temperature using cooling solution**.

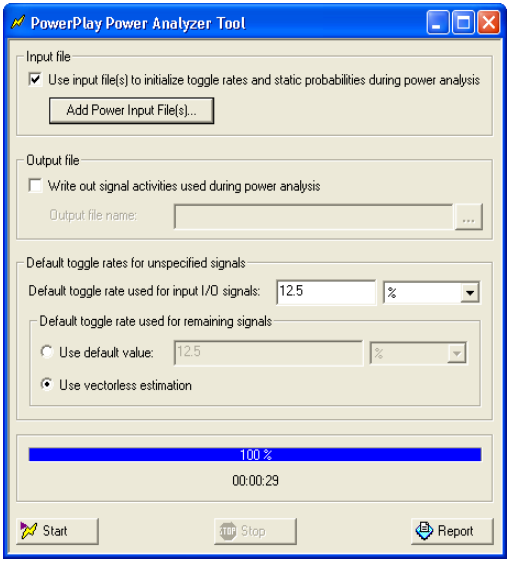
25. (Optional) Under **Board thermal modeling**, select the **Board thermal model** and type the **Board temperature**. This feature can only be turned on when you have selected **Auto compute junction temperature using cooling solution**.

    For more information about how to use the operating condition settings, refer to "Operating Conditions" on page 10–11.

26. Click **OK** to close the **Settings** dialog box.

27. On the Processing menu, click **PowerPlay Power Analyzer Tool**. The **PowerPlay Power Analyzer Tool** dialog box appears (Figure 10–20).

Figure 10–20. PowerPlay Power Analyzer Tool Dialog Box

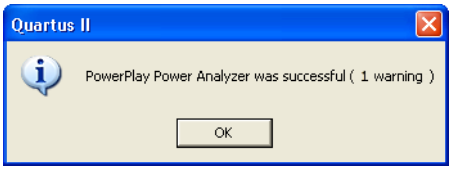


28. Click **Start** to run the PowerPlay Power Analyzer. Be sure that all the settings are correct.

    You can also make changes to some of your settings in this dialog box. For example, you can click the **Add Power Input File(s)** button to make changes to your input file(s).

29. After the PowerPlay Power Analyzer runs successfully, a message appears (Figure 10–21).

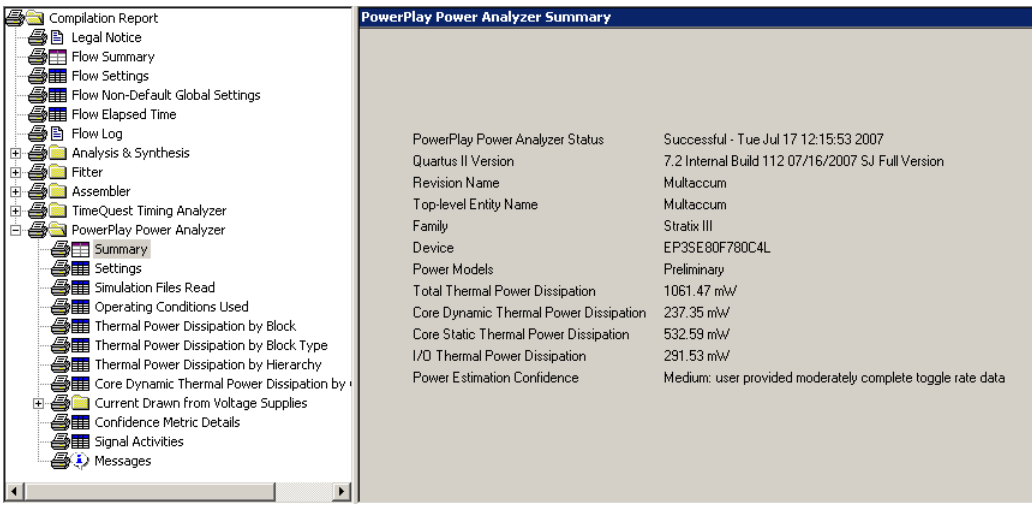
Figure 10–21. PowerPlay Power Analyzer Message



30. Click **OK**.

31. In the **PowerPlay Power Analyzer Tool** dialog box, click **Report** to open the PowerPlay Power Analyzer Summary window. You can also view the summary in the **PowerPlay Power Analyzer Summary** page of the **Compilation Report** (Figure 10–22).

Figure 10–22. PowerPlay Power Analyzer Summary



PowerPlay Power Analyzer Compilation Report

The PowerPlay Power Analyzer section of the Compilation Report is divided into the following sections.

Summary

This section of the report shows your design's estimated total thermal power consumption. This includes dynamic, static, and I/O thermal power consumption. The report also includes a confidence metric that reflects the overall quality of the data sources for the signal activities.

Settings

This section of the report shows your design's PowerPlay Power Analyzer settings information. This includes default input toggle rates, operating conditions, and other relevant setting information.

Simulation Files Read

This section of the report lists simulation output files (VCD file or SAF) used for power estimation.

Operating Conditions Used

This section of the report shows device characteristics, voltages, temperature, and cooling solution, if any, that were used during the power estimation. It also shows the entered junction temperature or auto-computed junction temperature that was used during the power analysis. This page is created only for Arria GX, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families.

Thermal Power Dissipated by Block

This section of the report shows estimated thermal dynamic power and thermal static power consumption categorized by atoms. This information provides designers with an estimated power consumption for each atom in their design.

Thermal Power Dissipation by Block Type (Device Resource Type)

This section of the report shows the estimated thermal dynamic power and thermal static power consumption categorized by block types. This information is further categorized by estimated dynamic and static power that was used, as well as providing an average toggle rate by block type. Thermal power is the power dissipated as heat from the FPGA device.

Thermal Power Dissipation by Hierarchy

This section of the report shows an estimated thermal dynamic power and thermal static power consumption categorized by design hierarchy. This is further categorized by the dynamic and static power that was used by the blocks and routing within that hierarchy. This information is very useful in locating problem modules in your design.

Core Dynamic Thermal Power Dissipation by Clock Domain

This section of the report shows the estimated total core dynamic power dissipation by each clock domain. This provides designs with estimated power consumption for each clock domain in their design. If the clock frequency for a domain is unspecified by a constraint, the clock frequency is listed as "unspecified." For all the combinational logic, the clock domain is listed as no clock with 0 MHz.

Current Drawn from Voltage Supplies

This section of the report lists the current that was drawn from each voltage supply. The V_{CCIO} voltage supply is further categorized by I/O bank and by voltage. The minimum safe power supply size (current supply ability) is also listed for each supply voltage. This page is created only for Arria GX, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families.

Confidence Metric Details

The confidence metric indicates the quality of the signal toggle rate data used to compute a power estimate. The confidence metric is low if the signal toggle rate data comes from sources that are considered poor predictors of real signal toggle rates in the device during an operation. Toggle rate data that comes from simulation, or user-entered assignments on specific signals, or entities are considered reliable. Toggle rate data from default toggle rates (for example, 12.5% of the clock period) or vectorless estimation are considered relatively inaccurate. This section gives an overall confidence rating in the toggle rate data, from low to high. It also summarizes how many pins, registers, and combinational nodes obtained their toggle rates from each of simulation, user entry, vectorless estimation, or default toggle rate estimations. This detailed information can help you understand how to increase the confidence metric, letting you decide on your own confidence in the toggle rate data.

Signal Activities

This section lists toggle rate and static probabilities assumed by power analysis for all signals with fan-out and pins. The signal type is provided (Pin, Registered, or Combinational), as well as the data source for the toggle rate and static probability. By default, all signal activities are reported. This may be turned off on the **PowerPlay Power Analyzer Settings** page by turning off the **Write signal activities to report file** option. Turning this option off may be advisable for a large design because of the large number of signals present. You can use the Assignment Editor to specify that activities for individual nodes or entities are reported by assigning an on value to those nodes for the Power Report Signal Activities assignment.

Messages

This section lists any messages generated by the Quartus II software during the analysis.

Specific Rules for Reporting

In the Stratix GX device, the XGM II State Machine block is always used together with GXB transceivers, so its power is lumped into the power for the transceivers. Therefore, the power for the XGM II State Machine block is reported as 0 Watts.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```

The *Scripting Reference Manual* includes the same information in PDF format.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Running the PowerPlay Power Analyzer from the Command Line

The separate executable that can be used to run the PowerPlay Power Analyzer is `quartus_pow`. For a complete listing of all command line options supported by `quartus_pow`, type the following at a system command prompt:

`quartus_pow --help` or `quartus_sh --qhelp` ↵

The following is an example of using the `quartus_pow` executable with project **sample.qpf**:

- To instruct the PowerPlay Power Analyzer to generate a PowerPlay Early Power Estimator file, type the following at a system command prompt:

```
quartus_pow sample --output_epe=sample.csv ↵
```

- To instruct the PowerPlay Power Analyzer to generate a PowerPlay Early Power Estimator file without doing the power estimate, type the following command at a system command prompt:

```
quartus_pow sample --output_epe=sample.csv --estimate_power=off ↵
```

- To instruct the PowerPlay Power Analyzer to use a SAF as input (**sample.saf**), type the following at a system command prompt:

```
quartus_pow sample --input_saf=sample.saf ↵
```

- To instruct the PowerPlay Power Analyzer to use two VCD files as input (**sample1.vcd** and **sample2.vcd**), perform glitch filtering on the VCD file, and use a default input I/O toggle rate of 10,000 transitions/second, type the following at a system command prompt:

```
quartus_pow sample --input_vcd=sample1.vcd
--input_vcd=sample2.vcd --vcd_filter_glitches=on
--default_input_io_toggle_rate=10000transitions/s ↵
```

- To instruct the PowerPlay Power Analyzer to not use any input file, a default input I/O toggle rate of 60%, no vectorless estimation, and a default toggle rate of 20% on all remaining signals, type the following at a system command prompt:

```
quartus_pow sample --no_input_file --default_input_io_toggle_rate=60%
--use_vectorless_estimation=off --default_toggle_rate=20% ↵
```



There are no command line options to specify the information found on the **PowerPlay Power Analyzer Settings Operating Conditions** page. The easiest way to specify these options is to use the Quartus II GUI.

A report file, *<revision name>***.pow.rpt**, is created by the `quartus_pow` executable and saved in the main project directory. The report file contains the same information as described in the "PowerPlay Power Analyzer Compilation Report" on page 10–39.

Conclusion

PowerPlay power analysis tools are designed for accurate estimation of power consumption from early design concept through design implementation. Designers can use the PowerPlay Early Power Estimator to estimate power consumption during the design concept stage. Power estimations can be refined during design implementation using the Quartus II PowerPlay Power Analyzer feature. The Quartus II PowerPlay Power Analyzer produces detailed reports that you can use to optimize designs for lower power consumption and verify that the design is within your power budget.

Referenced Documents

This chapter references the following documents:

- *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Quartus II Simulator* chapter in volume 3 of the *Quartus II Handbook*
- *Section I. Simulation* in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 10–5 shows the revision history for this chapter.

Table 10–5. Document Revision History

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Updated Figures 10–4, 10–9, 10–10, 10–11, and 10–22. • Updated "Generating a SAF or VCD File Using the Quartus II Simulator" on page 10–24. • Updated "Generating a VCD File Using a Third-Party Simulator" on page 10–28.	Updated for the Quartus II software version 7.2.
May 2007 v7.1.0	• Updated procedures for "Generating a SAF or VCD File Using the Quartus II Simulator" on page 10–24. • Updated figures. • Added "Document Revision History" on page 10–45.	Added support for Arria GX devices.
March 2007 v7.0.0	Added Cyclone III to list of devices supported (page 10-2)	—
November 2006 v6.1.0	• Updated "Generating a SAF or VCD File Using the Quartus II Simulator" by changing steps in certain processes to accommodate new functionality. • Updated "Operating Conditions" by adding Selectable Core Voltage option. • Updated Figure 10-2, 10-9, 10-10, 10-12, 10-14, 10-18, and 10-19.	Figure changes were made to accommodate the changes to the GUI. Also, added information for Stratix III devices.
May 2006 v6.0.0	Chapter title changed to *PowerPlay Power Analysis*. Updated for the Quartus II software version 6.0.0: • Added information about the EPE tools. • Added information about the power analyzer.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
May 2005 v5.0.0	• Updated information. • Updated figures. • New functionality for Quartus II software 5.0.	—
December 2004 v1.0	Initial release.	—

Section IV. Signal Integrity

As FPGA usage expands into more high-speed applications, signal integrity becomes an increasingly important factor to consider for an FPGA design.

Signal integrity issues must be taken into account as part of FPGA I/O planning and assignments, as well as in the design and layout of the printed circuit board (PCB) that must support the FPGA. Early design simulation is essential for preventing issues that may require a board redesign. The Quartus II software provides a number of features that will help you make smart board design decisions to ensure good signal integrity on all your high-speed interfaces.

This section includes the following chapter:

- Chapter 11, Signal Integrity Analysis with Third-Party Tools



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

11. Signal Integrity Analysis with Third-Party Tools

QII53020-7.2.0

Introduction

As FPGA devices are used in more high-speed applications, signal integrity and timing margin between the FPGA and other devices on the printed circuit board (PCB) become increasingly important considerations to ensure proper system operation. To avoid time consuming redesigns and expensive board respins, the topology and routing of critical signals must be simulated. The high-speed interfaces available on current FPGA devices must be modeled accurately and integrated into timing models and board-level signal integrity simulations. To do this, the tools used in the design of an FPGA and its integration into a PCB must be "board-aware," able to take into account properties of the board routing as well as the connected devices on the board.

The Quartus® II software provides a number of methodologies, resources, and tools to assist in ensuring good signal integrity and timing margin between an Altera® FPGA device and other components on the board. Three types of analysis are possible with the Quartus II software:

- I/O timing with a default or user-specified capacitive load and no signal integrity analysis (default)
- The Quartus II Advanced I/O Timing option utilizing a user-defined board trace model to produce enhanced timing reports from accurate "board-aware" simulation models
- Full board routing simulation in third-party tools using Altera provided or generated IBIS or HSPICE I/O models

I/O timing using a specified capacitive test load requires no special configuration other than setting the size of the load. I/O timing reports from Quartus II TimeQuest or the Quartus II Classic Timing Analyzer are generated based only on point-to-point delays within the I/O buffer and assume the presence of the capacitive test load with no other details about the board specified. The default size of the load is based on the I/O standard selected for the pin. Timing is measured to the FPGA pin with no signal integrity analysis details.

The Advanced I/O Timing option expands the details in I/O timing reports by taking board topology and termination components into account. A complete point-to-point board trace model is defined and accounted for in the timing analysis. This ability to define a board trace model is an example of how the Quartus II software is "board-aware."

In this case, timing and signal integrity metrics between the I/O buffer and the defined far end load are analyzed and reported in enhanced reports generated by the Quartus II TimeQuest Timing Analyzer.



For more information about defining capacitive test loads or how to use the Advanced I/O Timing option to configure a board trace model, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

This chapter focuses on the third type of analysis. The Quartus II software can export accurate HSPICE models with the built-in HSPICE Writer. You can run signal integrity simulations with these complete HSPICE models in Synopsys HSPICE. Input/Output Buffer Information Specification (IBIS) models of the FPGA I/O buffers are also created easily with the Quartus II IBIS Writer. You can integrate IBIS models into any third-party simulation tool that supports them, such as Mentor Graphics Hyperlynx software. With the ability to create industry-standard model definition files quickly, you can build accurate simulations that can provide data to help improve board-level signal integrity.

This chapter describes some of the basics of board-level signal integrity and why it should be taken into consideration as part of the general FPGA design flow. You will see that it is easy to produce accurate I/O models in the Quartus II software that take into account the unique properties of timing and signal integrity found in FPGA devices. You will learn how to add these models to your board routing simulations in the most widely used third-party simulation tools. Finally, you will find out where to go for more information about board-level signal integrity and how the Quartus II software and Altera FPGA devices fit into an overall high-speed system design.

This chapter is intended for FPGA and board designers. FPGA designers will learn about the concepts and steps involved in getting their designs simulated and how to adjust their designs to improve board-level timing and signal integrity. Board designers will learn how to get accurate models from the Quartus II software and how to use those models in their simulation software. To get the most out of this chapter, you should be familiar with the use of the Quartus II software. It is also helpful if you are familiar with some of the basic concepts involved in signal integrity and the design techniques and components required to have good signal integrity on a PCB. Finally, you should know how to set up simulations and use your selected third-party simulation tool. This chapter gives a basic overview of how to use the output from the IBIS Writer and HSPICE Writer in these tools, but it does not provide detailed instructions on their use.



For information about basic signal integrity concepts and signal integrity details pertaining to Altera FPGA devices, refer to the Altera Signal Integrity Center.

The Need for FPGA to Board Signal Integrity Analysis

When creating an FPGA design, the designer usually focuses on the FPGA logic design and functionality. A main focus for the design of the PCB to support the FPGA is to make sure FPGA I/O assignments match the correct pads and routing to ensure the FPGA signals are correctly connected to the rest of the circuit. In the past, this was all that was necessary to ensure proper operation. However, FPGA devices can now be configured with a wide assortment of high-speed interfaces that communicate with many other devices on the board.

With the introduction of high-speed interfaces to traditional FPGA design, it becomes necessary to make sure that timing and signal integrity margins between the FPGA and other devices on the board are within specification and tolerance before a single PCB is built. If the board trace is designed poorly or the route is too heavily loaded, noise in the signal can cause data corruption, while overshoot and undershoot can potentially damage input buffers over time if allowed to continue.

The use of the I/O model creation and analysis tools available in the Quartus II software early in the design process can help prevent problems before a costly board respin is needed. In general, creating and running accurate simulations is difficult and time consuming. The tools in the Quartus II software help by automating the I/O model setup and creation process by configuring the models specifically for your design. You will be able to set up and run accurate simulations quickly and acquire data that helps guide your FPGA and board design, using either the Advanced I/O Timing feature for analysis in the Quartus II software environment or the output from the IBIS and HSPICE Writers in third-party simulation tools.



The discussion of signal integrity in this chapter refers to board-level signal integrity based on I/O buffer configuration and board parameters, not simultaneous switching noise (SSN), also known as ground bounce or V_{CC} sag. SSN is a product of multiple output drivers switching at the same time, causing an overall drop in the voltage of the chip's power supply. This can cause temporary glitches in the specified level of ground or V_{CC} for the device. For a more thorough discussion of SSN and ways to prevent it, refer to application note AN *315: Guidelines for Designing High-Speed FPGA PCBs*.

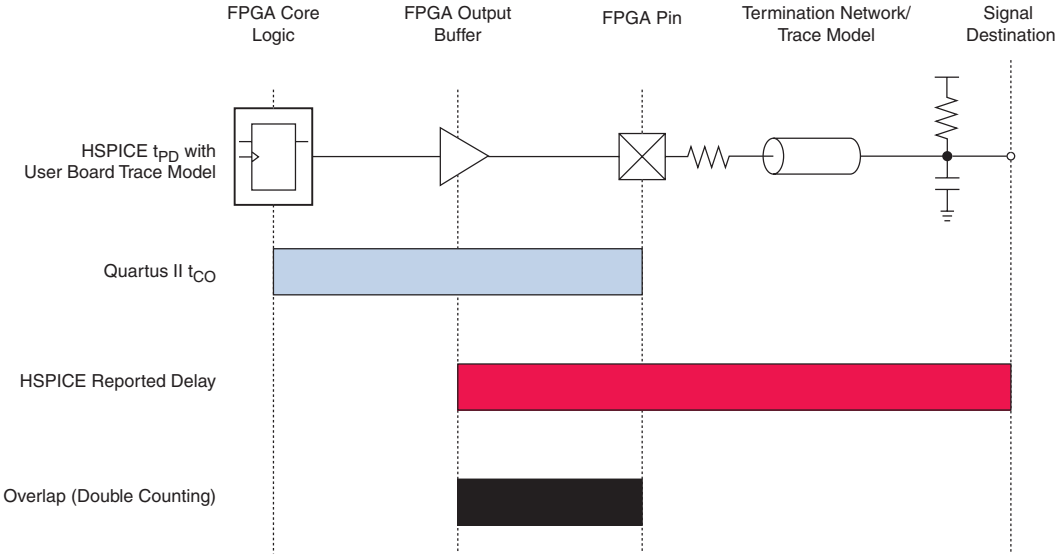
The Double Counting Problem for FPGA Output Timing

Simulating I/Os using accurate models is extremely helpful for finding and fixing FPGA I/O timing and board signal integrity issues before any boards are built. However, the usefulness of such simulations is directly related to the accuracy of the models used and whether the simulations are set up and performed correctly. To ensure accuracy in models and simulations created for FPGA output signals, the timing hand-off between t_{CO} timing in the Quartus II software and simulation-based board delay must be taken into account. If this hand-off is not handled correctly, the calculated delay could either count some of the delay twice or even miss counting some of the delay entirely.

Defining the Double Counting Problem

The double counting problem is inherent to the way output timing is analyzed versus the method used for HSPICE models. The timing analyzer tools in the Quartus II software measure delay timing for an output signal from the core logic of the FPGA design through the output buffer ending at the FPGA pin with a default capacitive load or a specified value for the selected I/O standard. This measurement is the t_{CO} timing variable as shown in Figure 11–1.

Figure 11–1. Double Counting Problem



HSPICE models for board simulation measure t_{PD} (propagation delay) from an arbitrary reference point in the output buffer, through the device pin, out along the board routing, and ending at the signal destination (the red bar in Figure 11–1).

It is immediately apparent that if these two delays were simply added together, the delay between the output buffer and the device pin would be counted twice in the calculation (the black bar in Figure 11–1). A model or simulation that does not account for this double count would create overly pessimistic simulation results, since the double counted delay can artificially limit I/O performance. To fix the problem, it may seem like simply subtracting the overlap between t_{CO} and t_{PD} would account for the double count. However, this adjustment would not be accurate because each measurement is based on a different load.

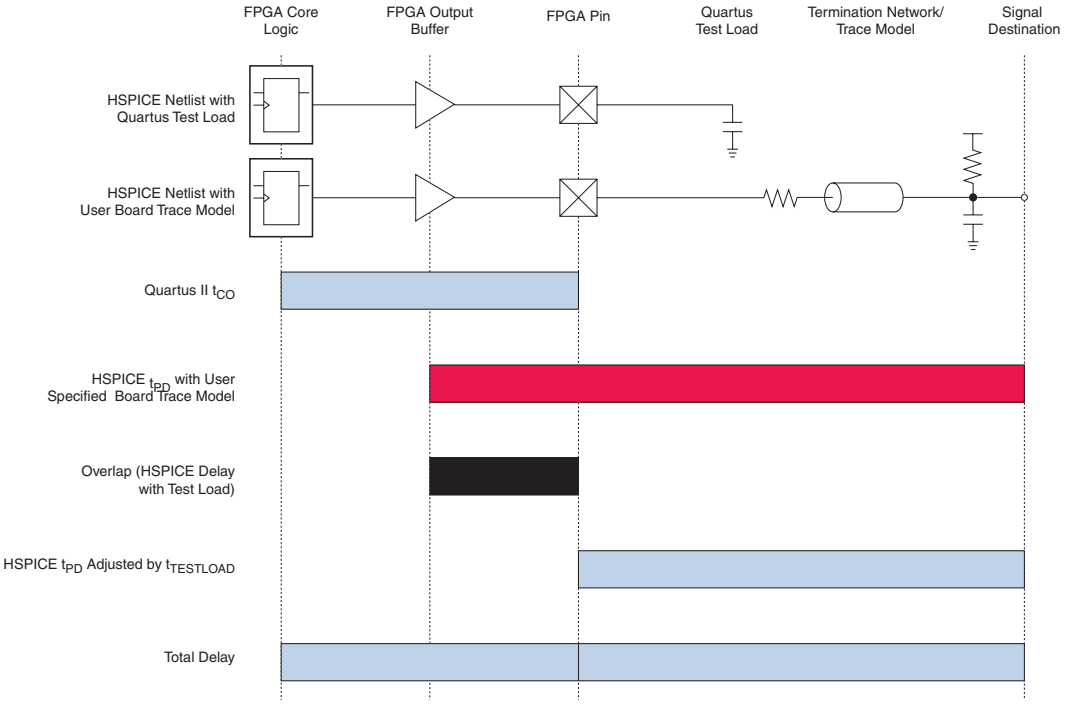


Input signals do not exhibit this problem because the HSPICE models for inputs stop at the FPGA pin instead of at the input buffer. In this case, simply adding the delays together produces an accurate measurement of delay timing.

The Solution to Double Counting

To adjust the measurements to account for the double counting, the delay between the arbitrary point in the output buffer selected by the HSPICE model and the FPGA pin must be subtracted from either t_{CO} or t_{PD} before adding the results together. The subtracted delay must also be based on a common load between the two measurements. This is done by repeating the HSPICE model measurement but with the same load used by the Quartus II software for the t_{CO} measurement. This second measurement, called $t_{TESTLOAD}$, is illustrated with the top circuit in Figure 11–2.

Figure 11–2. Common Test Loads Used for Output Timing



With $t_{TESTLOAD}$ known, the total delay for the output signal from the FPGA logic to the signal destination on the board, accounting for the double count, is calculated as shown in Equation 1.

$$t_{delay} = t_{CO} + (t_{PD} - t_{TESTLOAD}) \quad (1)$$

The preconfigured simulation files generated by the HSPICE Writer in the Quartus II software are designed to automatically account for the double counting problem based on this calculation. This makes it easy to perform accurate timing simulations without the need to manually make adjustments for double counting.

I/O Model Selection: IBIS or HSPICE

The Quartus II software can export two different types of I/O models that are useful for different simulation situations. IBIS models define the behavior of input or output buffers through the use of voltage-current (V-I) and voltage-time (V-t) data tables. HSPICE models, often referred to as HSPICE decks, include complete physical descriptions of the transistors and parasitic capacitances that make up an I/O buffer along with all the parameter settings needed to run a simulation. The HSPICE decks generated by the Quartus II software are preconfigured with the I/O standard, voltage, and pin loading settings for each pin in your design.

The choice of I/O model type is based on a number of factors. Table 11–1 provides a more detailed comparison of the two I/O model types as well as information and examples of situations about where and when they might be used.

Table 11–1. IBIS and HSPICE Model Comparison

Feature	IBIS Model	HSPICE Model
I/O Buffer Description	**Behavioral**—I/O buffers are described by voltage-current and voltage-time tables in typical, minimum, and maximum supply voltage cases.	**Physical**—I/O buffers and all components in a circuit are described by their physical properties, such as transistor characteristics and parasitic capacitances, as well as their connections to one another.
Model Customization	**Simple and limited**—The model completely describes the I/O buffer and does not usually need to be customized.	**Fully customizable**—Unless connected to an arbitrary board description, the description of the board trace model must be customized in the model file. All parameters of the simulation are also adjustable.
Simulation Set Up and Run Time	**Fast**—Simulations run quickly once set up correctly.	**Slow**—Simulations take time to set up and take longer to run and complete.
Simulation Accuracy	**Good**—For most simulations, accuracy is sufficient to make useful adjustments to the FPGA and/or board design to improve signal integrity.	**Excellent**—Simulations are highly accurate, making HSPICE simulation almost a requirement for any high-speed design where signal integrity and timing margins are tight.
Third-Party Tool Support	**Excellent**—Almost all third-party board simulation tools support IBIS.	**Good**—Most third-party tools that support SPICE support HSPICE. However, Synopsys HSPICE is required for simulations of Altera's encrypted HSPICE models.



For more information about IBIS files created by the Quartus II IBIS Writer and IBIS files in general, as well as links to websites with detailed information, refer to *AN 283: Simulating Altera Devices with IBIS Models*. For more information about HSPICE model files created by the Quartus II HSPICE Writer, refer to *AN 424: I/O Simulations Using HSPICE*.

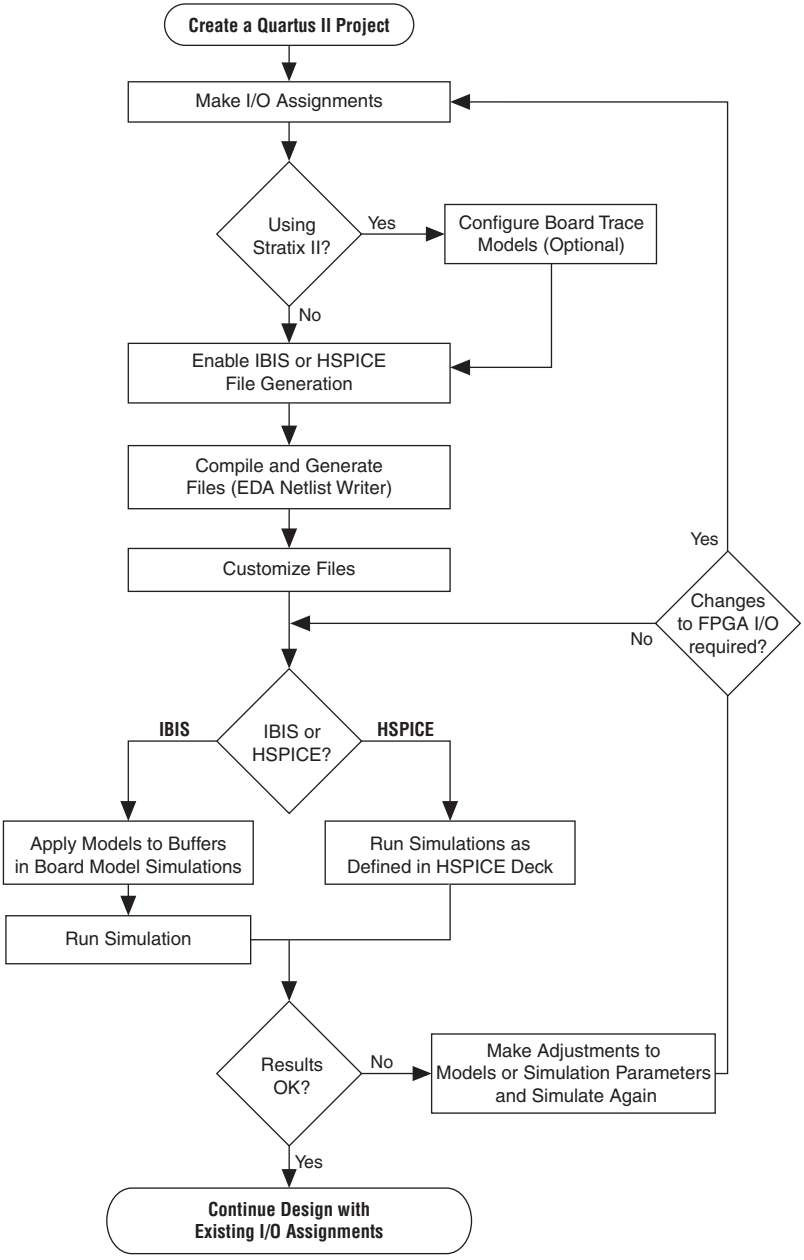
FPGA to Board Signal Integrity Analysis Flow

Board signal integrity analysis can take place at any point in the FPGA design process and is often performed both before and after board layout. If it is performed early in the process as part of a pre-PCB layout analysis, the models used for simulations can be more generic and can be changed as much as needed to see how adjustments improve timing or signal integrity and help with the design and routing of the PCB. Simulations and the resulting changes made at this stage allow you to analyze "what if" scenarios to better plan and implement your design. To assist with early board signal integrity analysis, you can download generic IBIS model files for each device family from the Altera website. If board signal integrity analysis is performed late in the design, it is typically used for a post-layout verification. The inputs and outputs of the FPGA are defined, and required board routing topologies and constraints are known. Simulations can help you find problems that may still exist in the FPGA or board design before fabrication and assembly. In either case, a simple process flow illustrates how to create accurate IBIS and HSPICE models from a design in the Quartus II software and transfer them to third-party simulation tools. Figure 11–3 shows this flow.



This chapter is organized around the type of model, IBIS or HSPICE, that you use for your simulations. Once you understand the steps in the analysis flow, refer to the section of this chapter that corresponds to the model type you are using.

Figure 11–3. Third-Party Board Signal Integrity Analysis Flow



Create I/O and Board Trace Model Assignments

If your design uses a Stratix II device, you can configure a board trace model for output signals or for bidirectional signals in output mode and automatically transfer its description to HSPICE decks generated by the HSPICE Writer. This helps improve simulation accuracy. To do this, turn on the **Enable Advanced I/O Timing** option in the **TimeQuest Timing Analyzer** page in the **Settings** dialog box and configure the board trace model assignment settings for each I/O standard used in your design. You can add series or parallel termination, specify the transmission line length, and set the value of the far-end capacitive load. You can configure these parameters in either the Board Trace Model view in the Pin Planner or by clicking **Device and Pin Options** in the **Device** page of the **Settings** dialog box.



For information about how to use Advanced I/O Timing and configure board trace models for the I/O standards used in your design, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

The Quartus II software can generate IBIS models and HSPICE decks without the need to configure a board trace model with the Advanced I/O Timing option. In fact, IBIS models ignore any board trace model settings other than the far-end capacitive load. If any load value is set other than the default, the delay given by IBIS models generated by the IBIS Writer cannot be used to account correctly for the double counting problem. The load value mismatch between the IBIS delay and the t_{CO} measurement of the Quartus II software prevents the delays from being safely added together. Warning messages displayed when the EDA Netlist Writer runs indicate when this mismatch occurs.

Enable Output File Generation

IBIS and HSPICE model files are not generated by the Quartus II software by default. To generate or update the files automatically during each project compilation, select the type of file to generate and a location where to save the file in the project settings. These settings can also be specified with commands in a Tcl script.

Generate the Output Files

The IBIS and HSPICE Writers in the Quartus II software are run as part of the EDA Netlist Writer during normal project compilation. If either writer is turned on in the project settings, IBIS or HSPICE files are created and stored in the specified location. For IBIS, a single file is generated containing information about all assigned pins, while HSPICE file generation creates separate files for each assigned pin. You can run the EDA Netlist Writer separately from a full compilation in the Quartus II

software or at the command line. However, you must fully compile the project or perform I/O Assignment Analysis at least once for the IBIS and HSPICE Writers to have information about the I/O assignments and settings in the design.

Customize the Output Files

The files generated by either the IBIS or HSPICE Writer are text files that you can edit and customize easily for design or experimentation purposes. IBIS files downloaded from the Altera website must be customized with the correct RLC values for the specific device package you have selected for your design. IBIS files generated by the IBIS Writer do not require this customization since they are automatically configured with the RLC values for your selected device. HSPICE decks require modification to include a detailed description of your board. With **Enable Advanced I/O Timing** turned on and a board trace model defined in the Quartus II software, generated HSPICE decks automatically include that model's parameters. However, it is recommended that you replace that model with a more detailed model that more accurately describes your board design. A default simulation included in the generated HSPICE decks measures delay between the FPGA and the far-end device. You can make additions or adjustments to the default simulation in the generated files to change the parameters of the default simulation or to perform additional measurements.

Set Up and Run Simulations in Third-Party Tools

Once you have generated the files, you can use them to perform simulations in your selected simulation tool. With IBIS models, you can apply them to input, output, or bidirectional buffer entities and quickly set up and run simulations. For HSPICE decks, the simulation parameters are included in the files. Open the files in Synopsys HSPICE and run simulations for each pin as needed. With HSPICE decks generated from the HSPICE Writer, the double counting problem is accounted for, ensuring that your simulations are accurate. Simulations that involve IBIS models created with anything other than the default loading settings in the Quartus II software must take the change in the size of the load between the IBIS delay and the Quartus II t_{CO} measurement into account. Warning messages during compilation alert you to this change.

Interpret Simulation Results

After running your simulations, you may find timing or signal integrity issues with your high-speed signals. Based on your simulation results, you can make adjustments to I/O assignment settings in the Quartus II software, such as drive strength or I/O standard, or make changes to your board routing or topology. After regenerating models in the Quartus II software based on the changes you have made, rerun the simulations to see if your changes corrected the problem.

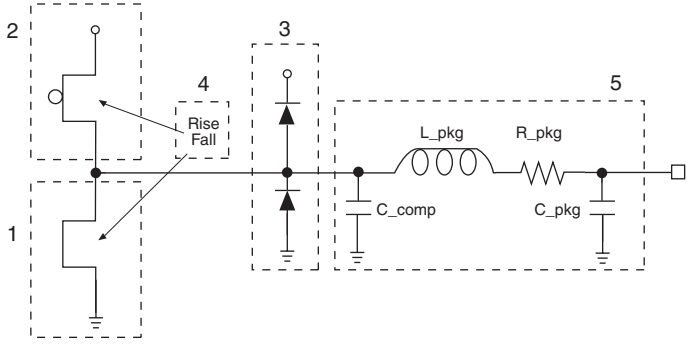
Simulation with IBIS Models

IBIS models provide a way to run accurate signal integrity simulations quickly. IBIS models describe the behavior of I/O buffers with voltage-current and voltage-time data curves. Because of their behavioral nature, IBIS models do not have to include any information about the internal circuit design of the I/O buffer. Most component manufacturers, including Altera, provide IBIS models for free download and use in signal integrity analysis simulation tools. You can download generic device family IBIS models from the Altera website for early design simulation or use the IBIS Writer to create custom IBIS models for your existing design.

Elements of an IBIS Model

An IBIS model file (**.ibs**) is a text file that describes the behavior of an I/O buffer across minimum, typical, and maximum temperature and voltage ranges with a specified test load. The tables and values specified in the IBIS file describe five basic elements of the I/O buffer. Figure 11–4 highlights each of these elements in the I/O buffer model.

Figure 11–4. Five Basic Elements in IBIS Models



The following elements correspond to each numbered block in Figure 11–4.

1. **Pulldown**—A voltage-current table describes the current when the buffer is driven low based on a pull-down voltage range of $-V_{CC}$ to $2V_{CC}$.

2. **Pullup**—A voltage-current table describes the current when the buffer is driven high based on a pull-up voltage range of $-V_{CC}$ to V_{CC}.

3. **Ground and Power Clamps**—Voltage-current tables describe the current when clamping diodes for electrostatic discharge (ESD) are present. The ground clamp voltage range is $-V_{CC}$ to V_{CC}, and the power clamp voltage range is $-V_{CC}$ to ground.

4. **Ramp and Rising/Falling Waveform**—A voltage-time (dv/dt) ratio describes the rise and fall time of the buffer during a logic transition. Optional rising and falling waveform tables can be added to more accurately describe the characteristics of the rising and falling transitions.

5. **Total Output Capacitance and Package RLC**—The total output capacitance includes the parasitic capacitances of the output pad, clamp diodes (if present), and input transistors. The package RLC is device package-specific and defines the resistance, inductance, and capacitance of the bond wire and pin of the I/O.

For more information about IBIS models and Altera-specific features, including links to the official IBIS specification, refer to *AN 283: Simulating Altera Devices with IBIS Models*.

Creating Accurate IBIS Models

There are two ways to obtain Altera device IBIS files for your board-level signal integrity simulations. You can download generic IBIS models from the Altera website or you can use the IBIS writer in the Quartus II software to create design-specific models.

Download IBIS Models

Altera provides IBIS models for almost all FPGA and FPGA configuration devices. Check the Download Center at **www.altera.com** to see if models for your selected device are available. You can use the IBIS models from the website to perform early simulations of the I/O buffers you expect to use in your design as part of a pre-layout analysis.

Downloaded IBIS models have the RLC package values set to one particular device in each device family. To accurately simulate your design with the model, you must adjust the RLC values in the IBIS model file to match the values for your particular device package by performing the following steps:

1. Download and expand the ZIP file (**.zip**) of the IBIS model for the device family you are using for your design. The **.zip** file contains the IBIS model file along with an IBIS model user guide and a model data correlation report.
2. Download the Package RLC Values spreadsheet for the same device family.
3. Open the spreadsheet and locate the row that describes the device package used in your design.
4. Copy the minimum, maximum, and typical values of resistance, inductance, and capacitance for your device package from the package's **I/O** row.
5. Open the IBIS model file in a text editor and locate the `[Package]` section of the file.
6. Overwrite the listed values copied with the values from the spreadsheet and save the file.

The IBIS model file is now customized for your device package and can be used for any simulation. IBIS models downloaded and used for simulations in this manner are generic. They describe only a certain set of models listed for each device on the IBIS model Download Center page on the Altera website. To create customized models for your design, use the IBIS Writer as described in the next section.

Generate Custom IBIS Models with the IBIS Writer

If you have started your FPGA design and have created custom I/O assignments, such as drive strength settings or the enabling of clamping diodes for ESD protection, you can use the Quartus II IBIS Writer to create custom IBIS models to more accurately reflect your assignments. IBIS models created with the IBIS Writer take I/O assignment settings into account.

If the **Enable Advanced I/O Timing** option is turned off, the generated IBIS model files are based on the load value setting for each I/O standard on the **Capacitive Loading** tab of the **Device and Pin Options** dialog box in the **Device** page of the **Settings** dialog box. With the **Enable Advanced**

I/O Timing option turned on, IBIS models use an effective capacitive load based on settings found in the board trace model on the **Board Trace Model** tab in the **Device and Pin Options** dialog box or the Board Trace Model view in the Pin Planner. The effective capacitive load is based on the sum of the **Near capacitance**, **Transmission line distributed capacitance**, and the **Far capacitance** settings in the board trace model. Resistances and transmission line inductance values are ignored.

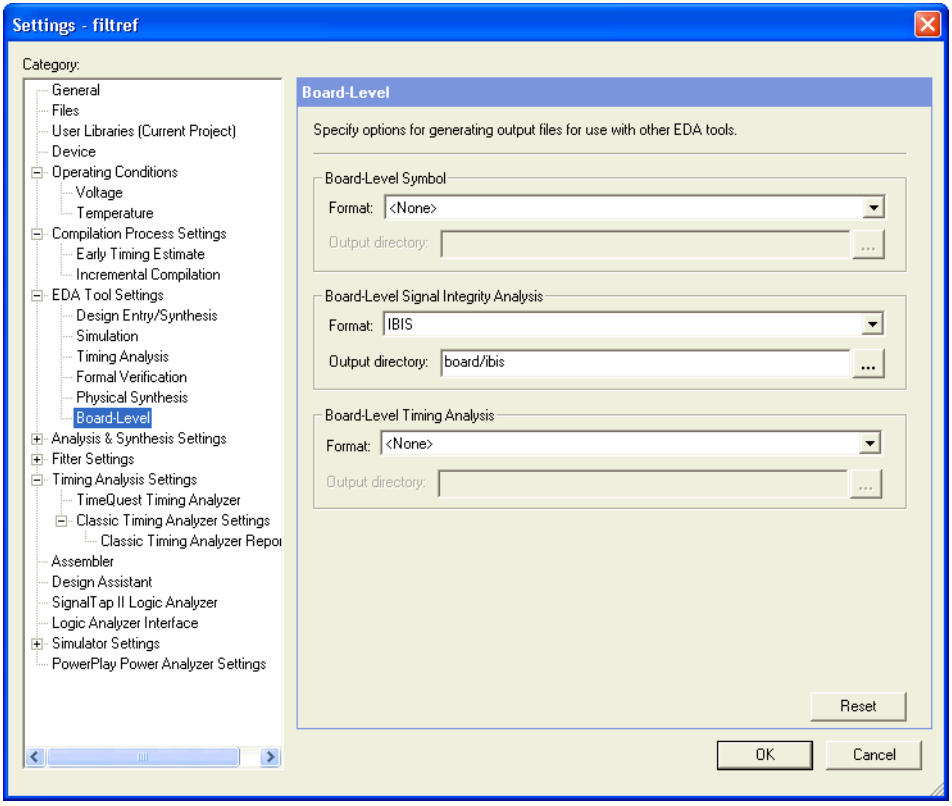


If any changes are made from the default load settings, the delay in the generated IBIS model cannot safely be added to the Quartus II t_{CO} measurement to account for the double counting problem. This is because the load values between the two delay measurements do not match. When this happens, the Quartus II software displays warning messages when the EDA Netlist Writer runs to remind you about the load value mismatch.

When the IBIS Writer is enabled, it generates a custom IBIS model file whenever the EDA Netlist Writer is run in the Quartus II software. To turn on the IBIS Writer and create custom IBIS model files, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, click the  icon to expand **EDA Tool Settings** and select **Board-Level**.
3. Under **Board-Level Signal Integrity Analysis Format**, in the **Format** list, select **IBIS** (Figure 11–5).

Figure 11–5. Enabling IBIS Model Generation in the Settings Dialog Box



4. IBIS models are stored in the <*project directory*>**/board/ibis** directory by default. To change the directory, click the browse button next to the **Output directory** box, and browse to the desired location.

5. Click **OK** to close the **Settings** dialog box.

6. If the project has not been compiled, run a full compilation to create a netlist and establish I/O assignments. On the Processing menu, click **Start Compilation**. The IBIS model file, named <*project name*>**.ibs**, is saved in the specified location.

7. If the project has been compiled before, you only need to run the EDA Netlist Writer to create or update the IBIS model file. On the Processing menu, point to Start and click **Start EDA Netlist Writer**. The IBIS model file is created or updated in the specified location.



You can save compilation time when creating the IBIS model file the first time for early design simulation by performing only required steps of the compilation process instead of a full compilation of your project. Run Analysis and Synthesis and I/O Assignment Analysis before creating the IBIS model file with the EDA Netlist Writer.



For more information about IBIS model generation, refer to the *AN 283: Simulating Altera Devices with IBIS Models* application note or the Quartus II Help.

Design Simulation Using the Mentor Graphics HyperLynx Software

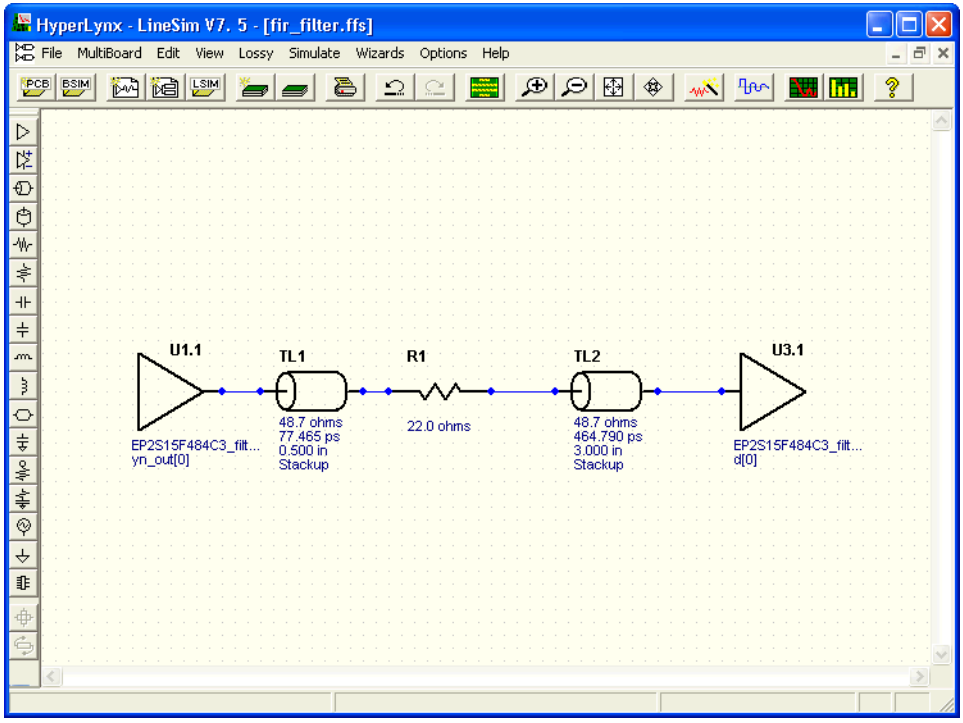
You must integrate IBIS models downloaded from the Altera website or created with the Quartus II IBIS Writer into board design simulations to accurately model timing and signal integrity. The HyperLynx software from Mentor Graphics is one of the most popular tools for design simulation. HyperLynx software makes it easy to integrate IBIS models into simulations.

The HyperLynx software is a PCB analysis and simulation tool for high-speed designs, consisting of two products, LineSim and BoardSim. LineSim is an early simulation tool. Before any board routing takes place, LineSim is used to simulate "what if" scenarios to assist in creating routing rules and defining board parameters. BoardSim is a post-layout tool used to analyze existing board routing. Specific nets are selected from a board layout file and simulated in a manner similar to LineSim. With board and routing parameters, and surrounding signal routing known, highly accurate simulations of the final fabricated PCB are possible. This section focuses on LineSim. Since the process of creating and running simulations is very similar for both LineSim and BoardSim, the details of IBIS model use in LineSim applies to simulations in BoardSim.

Simulations in LineSim are configured using a schematic GUI to create connections and topologies between I/O buffers, route trace segments, and termination components. LineSim provides two methods, cell-based and free-form, for creating routing schematics. Cell-based schematics are based on fixed cells consisting of typical placements of buffers, trace impedances, and components. Parts of the grid-based cells are filled with the desired objects to create the topology. A topology in a cell-based schematic is limited by the available connections within and between the cells.

A more robust and expandable way to create a circuit schematic for simulation is to use the free-form schematic format in LineSim as shown in Figure 11–6. The free-form schematic format makes it easy to place parts into any configuration and edit them as needed. This section describes the use of IBIS models with free-form schematics, but the process is nearly identical for cell-based schematics.

Figure 11–6. HyperLynx LineSim Free-Form Schematic Editor



When you use HyperLynx software to perform simulations, you typically perform the following steps:

1. Create a new LineSim free-form schematic document and set up the board stackup for your PCB using the Stackup Editor. In this editor, you specify board layer properties including layer thickness, dielectric constant, and trace width.

2. Create a circuit schematic for the net you want to simulate. The schematic represents all the parts of the routed net including source and destination I/O buffers, termination components, transmission line segments, and representations of impedance discontinuities such as vias or connectors.

3. Assign IBIS models to the source and destination I/O buffers to represent their behavior during operation.

4. Attach probes from the digital oscilloscope that is built in to LineSim to points in the circuit that you want to monitor during simulation. Typically, at least one probe is attached to the pin of a destination I/O buffer. For differential signals, you can attach a differential probe to both the positive and negative pins at the destination.

5. Configure and run the simulation. You can simulate a rising or falling edge and test the circuit under different drive strength conditions.

6. Interpret the results and make adjustments. Based on the waveforms captured in the digital oscilloscope, you can adjust anything in the circuit schematic to correct any signal integrity issues, such as overshoot or ringing. If necessary, you can make I/O assignment changes in the Quartus II software, regenerate the IBIS file with the IBIS Writer, and apply the updated IBIS model to the buffers in your HyperLynx software schematic.

7. Repeat the simulations and circuit adjustments until you are satisfied with the results. Once the operation of the net meets your design requirements, implement changes to your I/O assignments in the Quartus II software and/or adjust your board routing constraints, component values, and placement to match the simulation.



For more information about HyperLynx software, including schematic creation, simulation setup, model usage, product support, licensing, and training, refer to HyperLynx Help or the Mentor Graphics website at **www.mentor.com**.

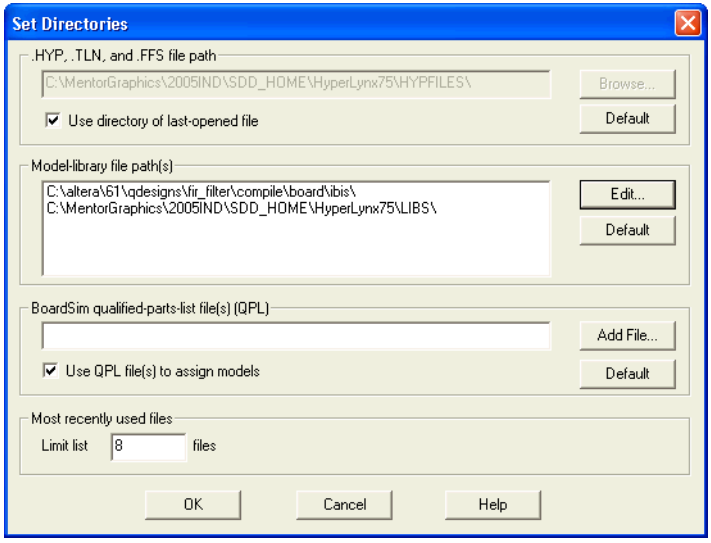
Configuring LineSim to Use Altera IBIS Models

You must configure LineSim to find and use the downloaded or generated IBIS models for your design. To do this, you add the location of your IBIS model file(s) to the LineSim Model Library search path. Then you apply a selected model to a buffer in your schematic.

To add the Quartus II software's default IBIS model location, *<project directory>***/board/ibis**, to the HyperLynx LineSim model library search path, perform the following steps in LineSim:

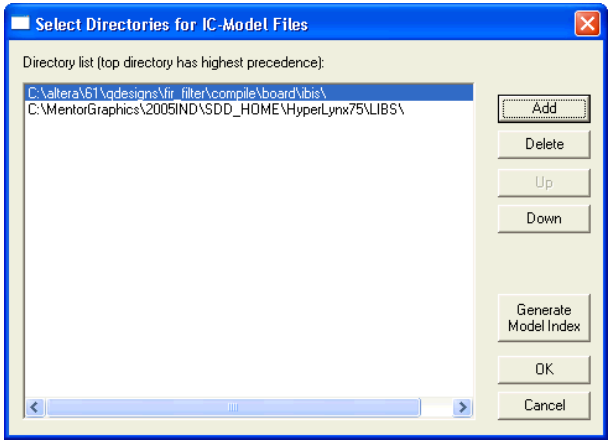
1. From the Options menu, click **Directories**. The **Set Directories** dialog box appears (Figure 11–7). The Model-library file path(s) list displays the order in which LineSim searches file directories for model files.

Figure 11–7. LineSim Set Directories Dialog Box



2. Click **Edit**. A dialog box appears where you can add directories and adjust the order in which LineSim searches them (Figure 11–8).

Figure 11–8. LineSim Select Directories Dialog Box



3. Click Add and browse to the default IBIS model location, *<project directory>*/**board/ibis**. Click **OK**.

4. Click **Up** to move the IBIS model directory to the top of the list, and click **Generate Model Index** to update LineSim's model database with the models found in the added directory.

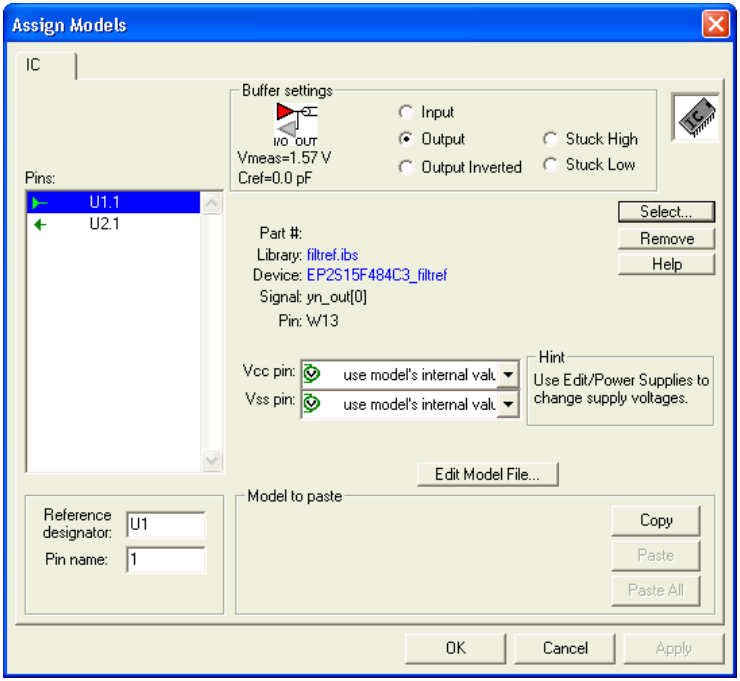
5. Click **OK**. The IBIS model directory for your project is added to the top of the Model-library file path(s) list. Click **OK** to close the Set Directories dialog box.

Integrating Altera IBIS Models into LineSim Simulations

Once the location for IBIS files is set, you can assign the downloaded or generated IBIS models to the buffers in your schematic. To do this, perform the following steps:

1. Double-click a buffer symbol in your schematic to open the **Assign Models** dialog box (Figure 11–9). You can also click **Assign Models** from the buffer symbol's right-click menu.

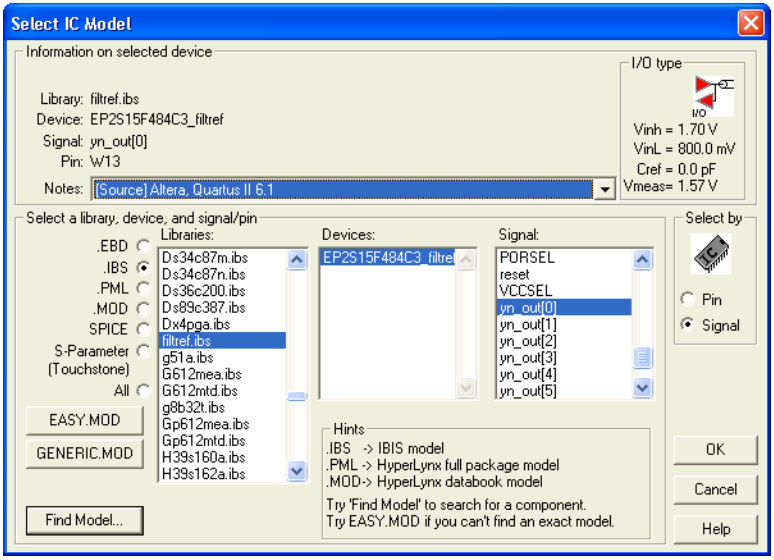
Figure 11–9. LineSim Assign Model Dialog Box



2. The pin of the buffer symbol you selected should be highlighted in the **Pins** list. If you want to assign a model to a different symbol or pin, select it from the list.

3. Click **Select**. The **Select IC Model** dialog box appears (Figure 11–10).

Figure 11–10. LineSim Select IC Model Dialog Box



4. To filter the list of available libraries to display only IBIS models, select **.IBS**. Scroll through the **Libraries** list, and click the name of the library for your design. By default, this is <*project name*>**.ibs**.

5. The device for your design should be selected as the only item in the **Devices** list. If not, select your device from the list.

6. From the **Signal** list, select the name of the signal you want to simulate. You can also choose to select by device pin number.

7. Click **OK**. The **Assign Models** dialog box displays the selected IBIS model file and signal.

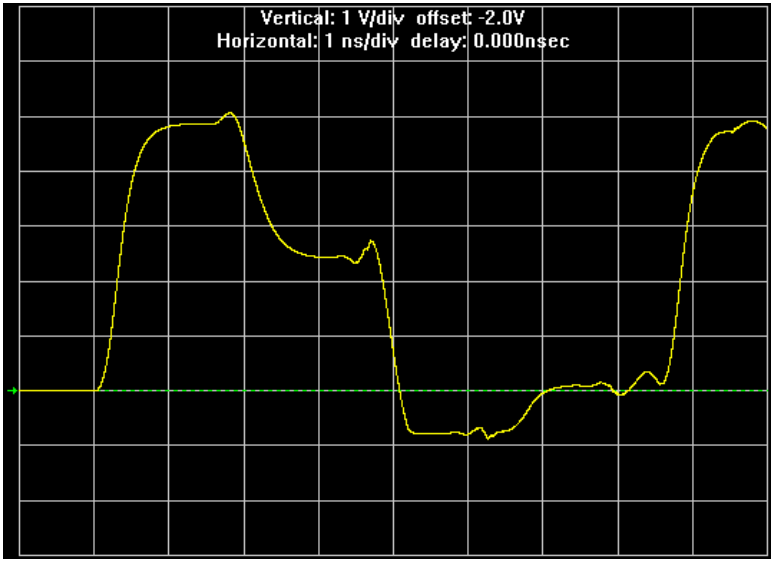
8. If applicable to the signal you chose, adjust the buffer settings as needed for the simulation.

9. Select and configure other buffer pins from the **Pins** list in the same manner. Click **OK** when all I/O models are assigned.

Running and Interpreting LineSim Simulations

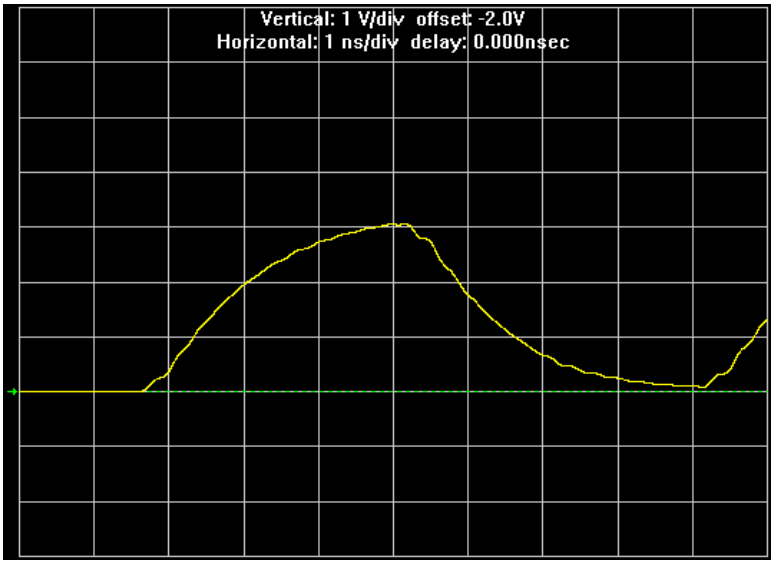
You can now run any desired simulations and make adjustments to the I/O assignments or simulation parameters as needed. For example, if after running a simulation you see too much overshoot in the simulated signal at the destination buffer as seen in Figure 11–11, you could adjust the drive strength I/O assignment setting to a lower value. Regenerate the IBIS model file, and run the simulation again to verify if the change fixed the problem.

Figure 11–11. Example of Overshoot in HyperLynx with IBIS Models



If you see a discontinuity or other anomalies at the destination, such as slow rise and fall times as shown in Figure 11–12, adjust the termination scheme or termination component values. After making these changes, rerun the simulation to check whether your adjustments solved the problem. In this case, it is not necessary to regenerate the IBIS model file.

Figure 11–12. Example of Signal Integrity Anomaly in HyperLynx with IBIS Models



For more information about board-level signal integrity and to learn about ways to improve it with simple changes to your design, visit the Altera Signal Integrity Center at **www.altera.com**.

Simulation with HSPICE Models

HSPICE decks are used to perform highly accurate simulations by precisely describing the physical properties of all aspects of a circuit. HSPICE decks describe I/O buffers, board components, and all the connections between them, as well as defining the parameters of the simulation to be run. By their nature, HSPICE decks are highly customizable and require a detailed description of the circuit under simulation to be effective. For Stratix II devices, when **Enable Advanced I/O Timing** is turned on, the HSPICE decks generated by the Quartus II HSPICE Writer automatically include board components and topology defined in the Board Trace Model that you configure in the Pin Planner or in the **Board Trace Model** tab of the **Device and Pin Options** dialog box. All HSPICE decks generated by the Quartus II software include compensation for the double count problem (for more information about the double count problem, refer to "The Double Counting Problem for FPGA Output Timing" on page 11–4). You can simulate with the default simulation parameters built in to the generated HSPICE decks or make adjustments to customize your simulation.

For more detailed information about the HSPICE model files created by the Quartus II HSPICE Writer, refer to *AN 424: I/O Simulations Using HSPICE*.

Supported Devices and Signaling

The HSPICE Writer in the Quartus II software version 6.1 supports the devices and signaling defined in Table 11–2. Only Stratix II devices support the creation of a board trace model in the Quartus II software for automatic inclusion in an HSPICE deck. Other devices require the board description to be manually added to the HSPICE file.

Table 11–2. HSPICE Writer Device and Signaling Support

Device	Input	Output	Single-Ended	Differential	Automatic Board Trace Model Description
Stratix® II	✓	✓	✓	✓	✓
Stratix II GX (non-HSSI signals)	✓	✓	✓	✓	—
HardCopy® II	✓	✓	✓	✓	—

If you are using a Stratix II device for your design, you can turn on **Enable Advanced I/O Timing** and configure the board trace model for each I/O standard used in your design. The HSPICE files will include the board trace description you create in the Board Trace Model view in the Pin Planner or the **Board Trace Model** tab in **Device and Pin Options** dialog box.

For more information about Advanced I/O Timing and configuring board trace models for the I/O standards in your design, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Creating Accurate HSPICE Models

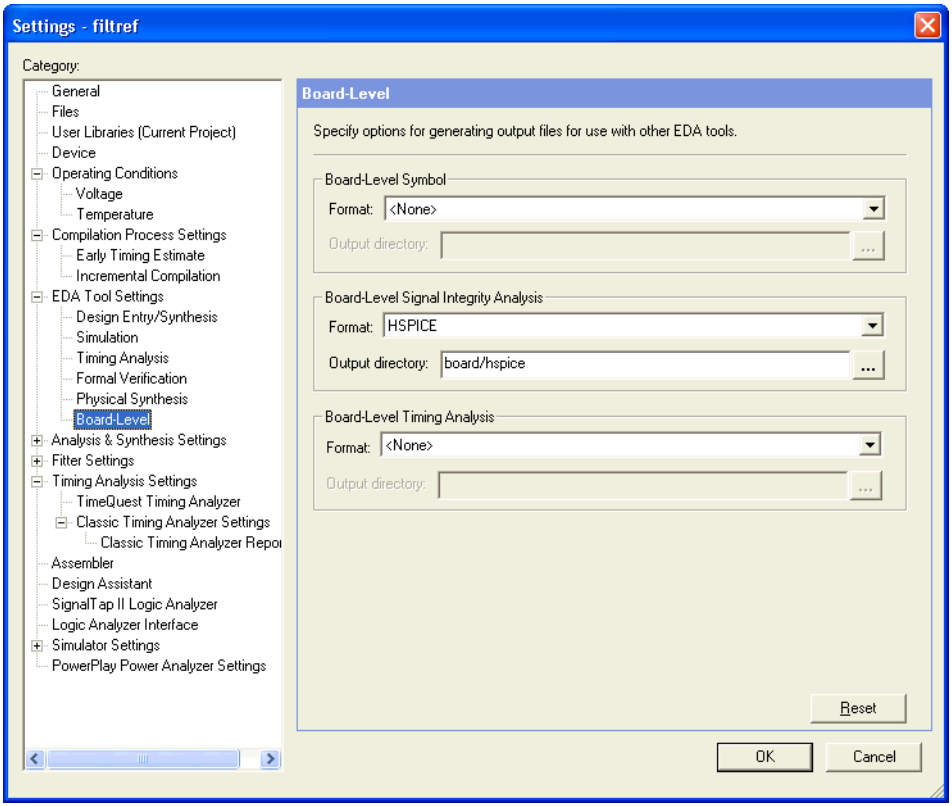
The HSPICE Writer must be turned on before HSPICE model files are created. HSPICE models are not generated by default in the Quartus II software. When enabled, the HSPICE Writer operates as part of the EDA Netlist Writer in the compilation process. When a project is fully compiled or the EDA Netlist Writer is run, the HSPICE Writer generates or updates the HSPICE model files.

Creating HSPICE Model Files Using the Quartus II GUI

To turn on the HSPICE Writer and create HSPICE deck files for each pin in your design, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, click the ⊞ icon to expand **EDA Tool Settings** and select **Board-Level**.
3. Under **Board-Level Signal Integrity Analysis Format**, in the **Format** list, select **HSPICE** (Figure 11–13).

Figure 11–13. Enabling HSPICE Deck and Model Generation in the Settings Dialog Box



4. HSPICE decks are stored in the <*project directory*>**/board/hspice** directory by default. To change the directory, click the browse button next to the **Output directory** box, and browse to the desired location.

5. Click **OK** to close the **Settings** dialog box.

6. If the project has not been compiled, run a full compilation to create a netlist and establish I/O assignments. On the Processing menu, click **Start Compilation**. HSPICE decks for each assigned pin, along with required model library subdirectories, are saved in the specified location.

7. If the project has been compiled, you only need to run the EDA Netlist Writer to create or update the HSPICE deck and model files. On the Processing menu, point to Start and click **Start EDA Netlist Writer**. The HSPICE decks and models are created or updated in the specified location.



You can save compilation time when creating HSPICE decks the first time for early design simulation by performing only required steps of the compilation process instead of a full compilation of your project. Run Analysis and Synthesis and I/O Assignment Analysis before creating the HSPICE deck files with the EDA Netlist Writer.

Preconfigured HSPICE simulation files generated by the HSPICE Writer are named <*device pin #*>_<*signal name*>_<*in*|*out*>**.sp**. Both an "in" and an "out" file are generated for bidirectional pins. HSPICE files are text files and can be edited with any ASCII text editor.

Two folders, named lib and cir, are also generated. These folders contain the encrypted I/O buffer descriptions and other information needed for running simulations. If you want to move the HSPICE model files to a different location, be sure to move these folders as well. The HSPICE model files include direct references to files in the lib and cir folders. If they are not in the same location, your HSPICE simulations will not run.

Creating HSPICE Model Files Using Tcl Scripting and the Command Line

If you use a script-based flow to compile your project, you can turn on the creation of HSPICE model files by including the following commands in your Tcl script (**.tcl** file):

```
set_global_assignment -name EDA_BOARD_DESIGN_SIGNAL_INTEGRITY_TOOL "HSPICE
(Signal Integrity)"
set_global_assignment -name EDA_OUTPUT_DATA_FORMAT HSPICE -section_id
eda_board_design_signal_integrity
```

```
set_global_assignment -name EDA_NETLIST_WRITER_OUTPUT_DIR <output_directory> -
section_id eda_board_design_signal_integrity
```

The <*output directory*> option specifies the location where HSPICE model files are saved. By default, the following directory is used:

<*project directory*>**/board/hspice**

You can run the HSPICE Writer at a command prompt by running the EDA Netlist Writer with the following command:

```
quartus_eda.exe <project name> --board_signal_integrity=on --format=HSPICE --
output_directory=<output directory>
```

The <*project name*> should match the name of the Quartus II Settings File (**.qsf**) for your project.

Customizing HSPICE Model Files

HSPICE models generated by the HSPICE Writer can be used for simulation as generated. A default board description is included, and a default simulation is set up to measure rise and fall delays for both input and output simulations which compensates for the double counting problem. However, Altera recommends that you customize the board description to more accurately represent your routing and termination scheme. To do this, open the generated HSPICE model files for all pins you want to simulate, and locate the following commented section:

```
*//////////////////////////////////////////////////////////////////////
* I/O Board Trace and Termination Description
* - Replace this with your board trace and termination description
*//////////////////////////////////////////////////////////////////////
```

Replace the board description in this section with a description of your board or the board topology you would like to simulate in each HSPICE file.

For input simulations, you must include a description of the device that provides the stimulus for the signal. Locate the following comments that indicate where to place the stimulus device description in the file:

```
*//////////////////////////////////////////////////////////////////////
* Sample source stimulus placeholder
*  - Replace this with your I/O driver model
*//////////////////////////////////////////////////////////////////////
```



For more information about configuring and customizing HSPICE model files for simulation, refer to the HSPICE manual.

Design Simulation Using Synopsys HSPICE

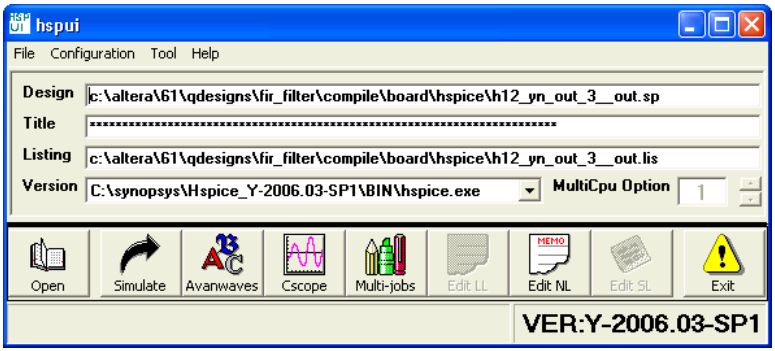
Synopsys HSPICE is an industry standard SPICE simulation tool; it is required for running SPICE simulation with Altera's encrypted HSPICE models. While you can use HSPICE model files in other tools, such as Mentor Graphics HyperLynx software, Synopsys HSPICE is still required to decrypt the models. You can use Synopsys HSPICE along with the included Avanwaves viewer to run simulations and view the results as waveforms.

For more information about Synopsys HSPICE, including licensing, installation, usage, support, and training, refer to the HSPICE manual or the Synopsys website at **www.synopsys.com**.

Running HSPICE Simulations

Since simulation parameters are configured directly in the HSPICE model files, running a simulation requires only that you open an HSPICE file in the HSPICE User Interface (hspui) and start the simulation. The hspui window is shown in Figure 11–14.

Figure 11–14. HSPICE hspui Window



Click **Open** and browse to the location of the HSPICE model files generated by the Quartus II HSPICE Writer. The default location for HSPICE model files is <*project directory*>**/board/hspice**. Select the **.sp** file, generated by the HSPICE Writer, for the signal you want to simulate and click **OK**.

Click **Simulate** to run the simulation. The status of the simulation is displayed in the window and saved in a **.lis** file with the same name as the **.sp** file when the simulation is complete. Check the **.lis** file if an error occurs during the simulation requiring a change in the **.sp** file to fix.

Viewing and Interpreting Tabular Simulation Results

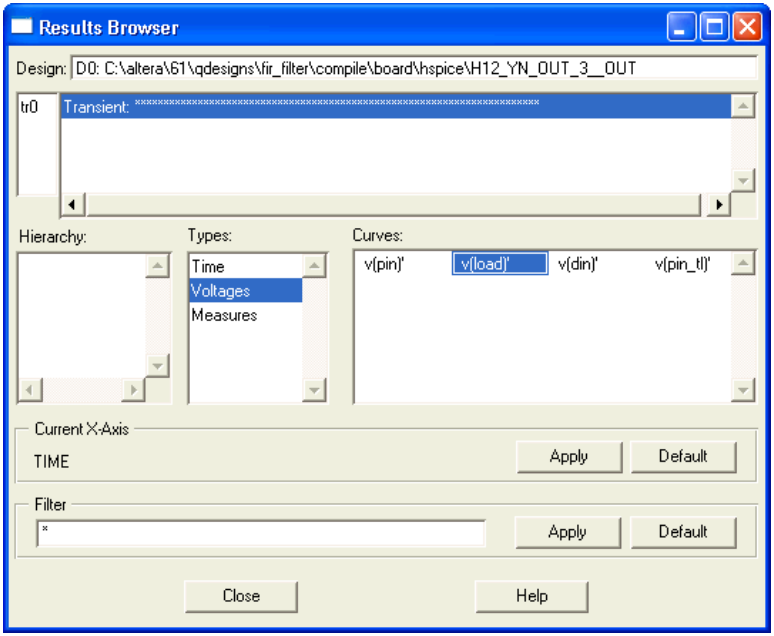
The **.lis** file stores the collected simulation data in tabular form. The default simulation configured by the HSPICE Writer produces delay measurements for rising and falling transitions on both input and output simulations. These measurements are found in the **.lis** file and named `tpd_rise` and `tpd_fall`. For output simulations, these values are already adjusted for the double count. Add either of these measurements to the Quartus II t_{CO} delay to determine the complete delay from the FPGA logic to the load pin. For input simulations, add either of these measurements to the Quartus II t_{SU} and t_H delay values to calculate the complete delay from the far end stimulus to the FPGA logic. Other values found in the **.lis** file, such as `tpd_uncomp_rise`, `tpd_uncomp_fall`, `t_dblcnt_rise`, and `t_dblcnt_fall`, are parts of the double count compensation calculation. These values are not needed for further analysis.

Viewing Graphical Simulation Results

You can quickly view the results of the simulation as a graphical waveform display using the Avanwaves viewer included with HSPICE. With the default simulation configured by the HSPICE Writer, you can view the simulated waveforms at both the source and destination in input and output simulations.

To see the waveforms for the simulation, in the HSPICE hspui window, click **Avanwaves**. The Avanwaves viewer opens and displays the Results Browser as shown in Figure 11–15.

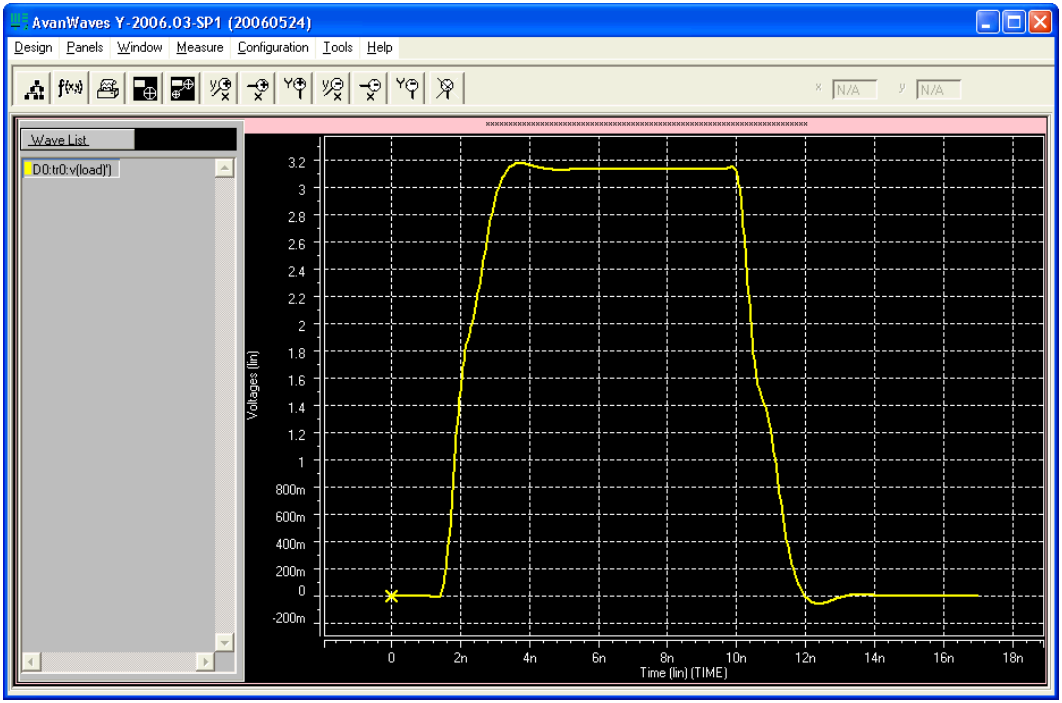
Figure 11–15. HSPICE Avanwaves Results Browser



The Results Browser lets you quickly select which waveform to view in the main viewing window. If multiple simulations are run on the same signal, the list at the top of the Results Browser displays the results of each simulation. Click the simulation description to select which simulation to view. By default, the descriptions are derived from the first line of the HSPICE file, so the description may appear as a line of asterisks.

Select the type of waveform to view. With the default simulation, select **Voltages** from the **Types** list to see the source and destination waveforms. On the **Curves** list, double-click the waveform you want to view. The waveform appears in the main viewing window. You can zoom in and out and adjust the view as desired (Figure 11–16).

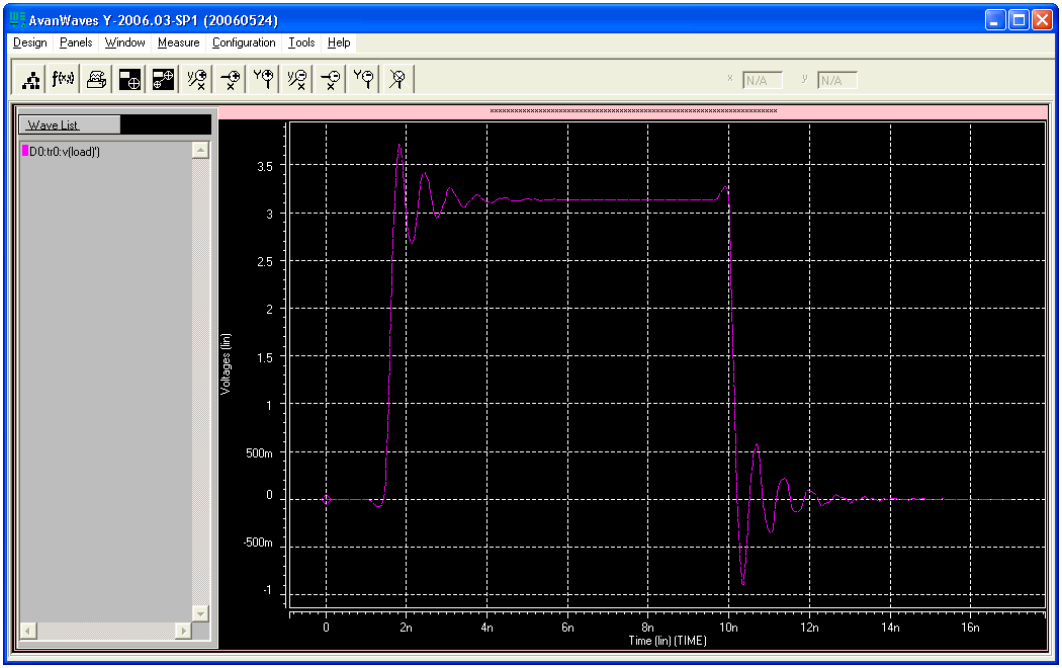
Figure 11–16. Avanwaves Waveform Viewer



Making Design Adjustments Based on HSPICE Simulations

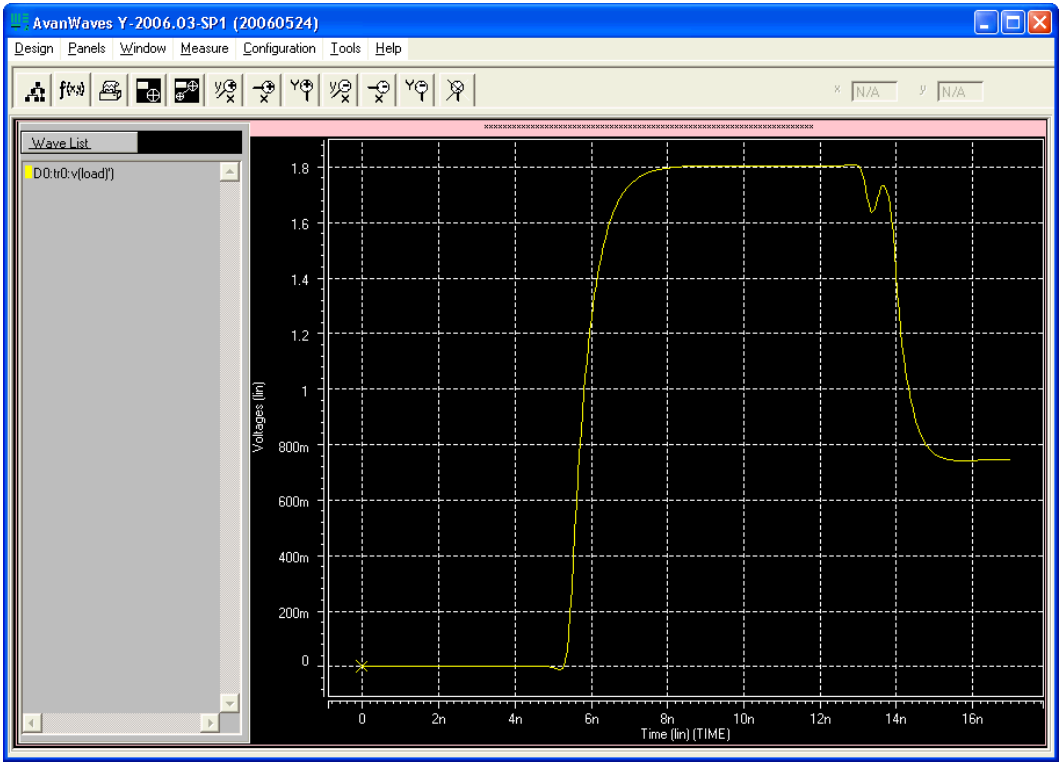
Based on the results of your simulations, you can make adjustments to the I/O assignments or simulation parameters if required. For example, after you run a simulation and see overshoot or ringing in the simulated signal at the destination buffer as shown in the example in Figure 11–17, you can adjust the drive strength I/O assignment setting to a lower value. Regenerate the HSPICE deck, and run the simulation again to verify that the change fixed the problem.

Figure 11–17. Example of Overshoot in the Avanwaves Waveform Viewer



If there is a discontinuity or any other anomalies at the destination as shown in the example in Figure 11–18, adjust the board description in the Quartus II Board Trace Model (for Stratix II devices) or in the generated HSPICE model files to change the termination scheme or adjust termination component values. After making these changes, regenerate the HSPICE files, if necessary, and rerun the simulation to verify whether your adjustments solved the problem.

Figure 11–18. Example of Signal Integrity Anomaly in the Avanwaves Waveform Viewer



For more information about board-level signal integrity and to learn about ways to improve it with simple changes to your FPGA design, refer to the Altera Signal Integrity Center.

Conclusion

As FPGA devices are used in more high-speed applications, it becomes increasingly necessary to perform board-level signal integrity analysis simulations. You must run such simulations to ensure good signal integrity between the FPGA and any connected devices. The Quartus II software helps to simplify this process with the ability to automatically generate I/O buffer description models easily with the IBIS and HSPICE Writers. IBIS models can be integrated into a third party signal integrity analysis workflow using a tool such as Mentor Graphics HyperLynx software, generating quick and accurate simulation results. HSPICE decks include preconfigured simulations and only require descriptions of board routing and stimulus models to create highly accurate simulation

results using Synopsys HSPICE. Either type of simulation helps prevent unnecessary board spins, increasing your productivity and decreasing your costs.

Referenced Documents

This chapter references the following documents:

- *AN 283: Simulating Altera Devices with IBIS Models*
- *AN 424: I/O Simulations Using HSPICE*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 11–3 shows the revision history for this chapter.

Table 11–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 11–36.	—
May 2007 v7.1.0	Added Referenced Documents.	—
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Initial Release	—

Section V. In-System Design Debugging

Debugging today's FPGA designs can be a daunting task. As your product requirements continue to increase in complexity, the time you spend on design verification continues to rise. To get your product to market as quickly as possible, you must minimize design verification time. To help alleviate the time-to-market pressure, you need a set of verification tools that are powerful, yet easy to use.

The Quartus® II software SignalTap® II Logic Analyzer and the SignalProbe™ features analyze internal device nodes and I/O pins while operating in-system and at system speeds. The SignalTap II Logic Analyzer uses an embedded logic analyzer to route the signal data through the JTAG port to either the SignalTap II Logic Analyzer or an external logic analyzer or oscilloscope. The SignalProbe feature uses incremental routing on unused device routing resources to route selected signals to an external logic analyzer or oscilloscope. A third Quartus II software feature, the Chip Editor, can be used in conjunction with the SignalTap II and SignalProbe debugging tools to speed up design verification and incrementally fix bugs uncovered during design verification. This section explains how to use each of these features.

This section includes the following chapters:

- Chapter 12, Quick Design Debugging Using SignalProbe
- Chapter 13, Design Debugging Using the SignalTap II Embedded Logic Analyzer
- Chapter 14, In-System Debugging Using External Logic Analyzers
- Chapter 15, In-System Updating of Memory and Constants
- Chapter 16, Design Debugging Using In-System Sources and Probes



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

12. Quick Design Debugging Using SignalProbe

QII53008-7.2.0

Introduction

Hardware verification can be a lengthy and expensive process. The SignalProbe incremental routing feature helps reduce the hardware verification process and time-to-market for system-on-a-programmable-chip (SOPC) designs.

Easy access to internal device signals is important in design or debugging. The SignalProbe feature makes design verification more efficient by quickly routing internal signals to I/O pins without affecting the design. Starting with a fully routed design, you can select and route signals for debugging to either previously reserved or currently unused I/O pins.

You can use the SignalProbe feature with the Stratix® series, Cyclone® series, MAX® II, and APEX™ series device families.

This chapter is divided into two sections. If you are using the SignalProbe feature to debug your Stratix series, Cyclone series, and MAX II device, then refer to "Debugging Using the SignalProbe Feature" on page 12–4. If you are using the SignalProbe feature to debug your APEX series device, refer to "Using SignalProbe with the APEX Device Family" on page 12–19.

On-Chip Debugging Tool Comparison

The Quartus® II software provides a number of different ways to help debug your FPGA design after programming the device. The SignalTap® II Logic Analyzer, SignalProbe, and the Logic Analyzer Interface (LAI) share some similar features, but each has advantages. In some debugging situations, it can be difficult to decide which tool is best to use or whether multiple tools are required. Table 12–1 compares common debugging features between these tools and provides suggestions for which is the best tool to use for a given feature.

Note that "✓" indicates the suggested best tool for the feature, "—" indicates that while the tool is available for that feature, that tool may not give the best results, and "N/A" indicates that the feature is not applicable for the selected tool.

Table 12–1. Suggested On-Chip Debugging Tools for Common Debugging Features Note (1) (Part 1 of 2)

Feature	SignalProbe	Logic Analyzer Interface (LAI)	SignalTap II Embedded Analyzer	Description
Large Sample Depth	N/A	✓	—	An external logic analyzer used with the LAI has a bigger buffer to store more captured data than the SignalTap II Logic Analyzer. No data is captured or stored with SignalProbe.
Ease in Debugging Timing Issue	N/A	✓	—	An external logic analyzer used with the LAI provides you with access to timing mode, enabling you to debug combined streams of data.
Minimal Effect on Logic Design	✓	✓ *(2)*	✓ *(2)*	SignalProbe incrementally routes nodes to pins, not affecting the design at all. The LAI adds minimal logic to a design, requiring fewer device resources. The SignalTap II Logic Analyzer has little effect on the design when it is set as a separate design partition using incremental compilation.
Short Compile and Recompile Time	✓	✓ *(2)*	✓ *(2)*	SignalProbe attaches incrementally routed signals to previously reserved pins, requiring very little recompilation time to make changes to source signal selections. The SignalTap II Logic Analyzer and the LAI can take advantage of incremental compilation to refit their own design partitions to decrease recompilation time.

Table 12–1. Suggested On-Chip Debugging Tools for Common Debugging Features Note (1) (Part 2 of 2)

Feature	SignalProbe	Logic Analyzer Interface (LAI)	SignalTap II Embedded Analyzer	Description
Triggering Capability	N/A	✓	—	Although advanced triggering is available in the SignalTap II Logic Analyzer, many additional triggering options are only available on an external logic analyzer when used with the LAI.
I/O Usage	—	—	✓	No additional output pins are required with the SignalTap II Logic Analyzer. Both the LAI and SignalProbe require I/O pin assignments.
Acquisition Speed	N/A	—	✓	The SignalTap II Logic Analyzer can acquire data at speeds of over 200 MHz. The same acquisition speeds are obtainable with an external logic analyzer used with the LAI, but signal integrity issues may limit this.
No JTAG Connection Required	✓	—	—	An FPGA design with the SignalTap II Logic Analyzer or the LAI requires an active JTAG connection to a host running the Quartus II software. SignalProbe does not require a host for debugging purposes.
External Equipment	—	—	✓	The SignalTap II Logic Analyzer logic is completely internal to the programmed FPGA device. No extra equipment is required other than a JTAG connection from a host running the Quartus II software or the stand-alone SignalTap II software. SignalProbe and the LAI require the use of external debugging equipment, such as multimeters, oscilloscopes, or logic analyzers.

Notes to Table 12–1:

(1) ✓ indicates the suggested best tool for the feature.
— indicates that while the tool is available for that feature, that tool may not give the best results.
N/A indicates that the feature is not applicable for the selected tool.

(2) When used with incremental compilation.

Debugging Using the SignalProbe Feature

The SignalProbe feature enables you to reserve available pins and route internal signals to those reserved pins, while preserving the behavior of your design. SignalProbe is an effective debugging tool providing visibility into your FPGA.



This section describes the SignalProbe process for the Stratix series, Cyclone series, and MAX II device families. Using SignalProbe with APEX devices is described in "Using SignalProbe with the APEX Device Family" on page 12–19. APEX devices do not support post-fit netlist changes made as engineering change orders (ECOs).

You can reserve pins for SignalProbe and assign I/O standards before or after a full compilation. Each SignalProbe source to SignalProbe pin connection is implemented as an ECO change that is applied to your netlist after a full compilation.

To route the internal signals to the device's reserved pins for SignalProbe, perform the following tasks:

1. Reserve the SignalProbe Pins, described on page 12–4.
2. Perform a Full Compilation, described on page 12–6.
3. Assign a SignalProbe Source, described on page 12–6.
4. Add Registers to the Pipeline Path to SignalProbe Pin, described on page 12–7.
5. Perform a SignalProbe Compilation, described on page 12–9.
6. Analyze the Results of the SignalProbe Compilation, described on page 12–10.
7. Generate the Programming File, described on page 12–11.

Reserve the SignalProbe Pins

You can reserve SignalProbe pins before or after compiling your design. Reserving SignalProbe pins before a compilation is optional. You can also reserve any unused I/Os of the device for SignalProbe pins after compilation. You can assign sources easily after reserving your SignalProbe pins. The sources for SignalProbe pins are the internal nodes and registers in the post-compilation netlist that you want to probe.

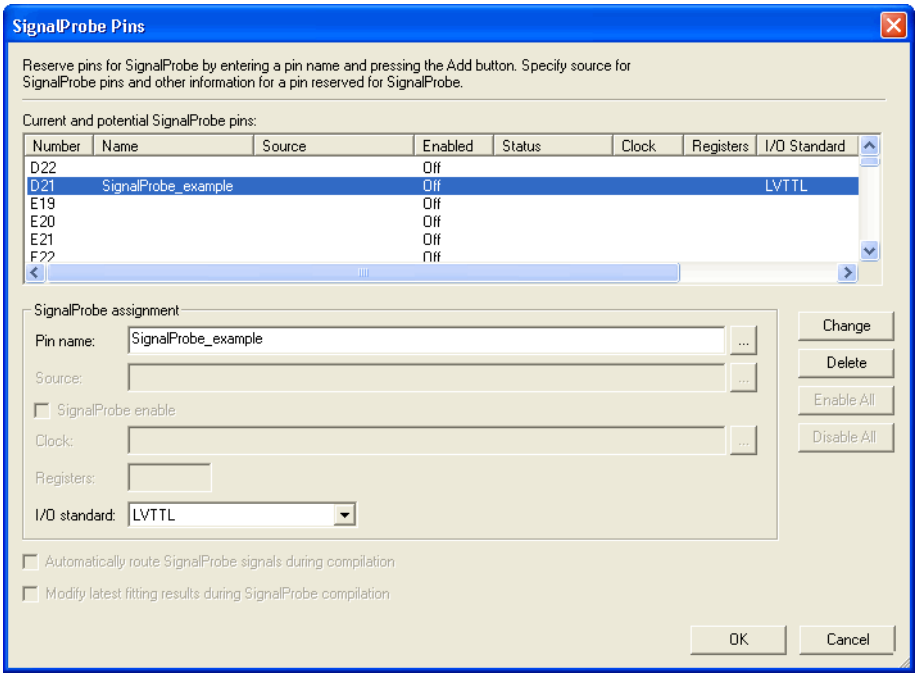


Although you can reserve SignalProbe pins using many features within the Quartus II software, including the Pin Planner and the Tcl interface, you should use the **SignalProbe Pins** dialog box to create and edit your SignalProbe pins.

To reserve an available package pin as a SignalProbe pin using the **SignalProbe Pins** dialog box, perform the following steps:

1. On the Tools menu, click **SignalProbe Pins**. The **SignalProbe Pins** dialog box appears (Figure 12–1). The SignalProbe pin name and I/O standard appear as the only fields that are editable if a place and route, or fit, has not been performed.

Figure 12–1. Reserving a SignalProbe Pin in the SignalProbe Pins Dialog Box



2. In the **Current and potential SignalProbe pins** list, click on a pin from the **Number** column and type your SignalProbe pin name into the **Pin name** box.

3. Select an I/O standard from the **I/O standard** drop-down list.

4. Click **Add** to add the new SignalProbe pin or **Change** if you are editing a previously reserved pin for SignalProbe. (Figure 12–1 shows the dialog box editing a previously reserved pin; if you were adding a new SignalProbe pin, the **Add** button appears instead of the **Change** button.)

5. Click **OK**.

Perform a Full Compilation

You must complete a full compilation to generate an internal netlist containing a list of internal nodes to probe to a SignalProbe outpin.

To perform a full compilation, on the processing menu, click **Start Compilation**.

Assign a SignalProbe Source

A SignalProbe source can be any combinational node, register, or pin in your post-compilation netlist. To find a SignalProbe source, use the SignalProbe filter in the Node Finder to filter out all sources that cannot be probed. You may not be able to find a particular internal node because the node may be optimized away during synthesis, or the node cannot be routed to the SignalProbe pin, as it is untappable. For example, internal nodes and registers within the Gigabit transceivers can not be probed because there are no physical routes to the pins available.



To probe virtual I/O pins generated in low-level partitions in an incremental compilation flow, select the source of the logic that feeds the Virtual Pin as your SignalProbe source pin.

To assign a SignalProbe source to your SignalProbe reserved pin, perform the following steps:

1. On the Tools menu, click **SignalProbe Pins**. The **SignalProbe Pins** dialog box appears (Figure 12–1 on page 12–5).

2. If a SignalProbe reserved pin is shown, click on the pin in the **Current and potential SignalProbe pins** list. Alternately, you can click on an available pin number in the **Current and potential SignalProbe pins** list and type a new SignalProbe pin name into the **Pin name** box.

3. In the **Source** box, specify the source name. Click the browse button. The **Node Finder** dialog box appears.

4. When you open the **Node Finder** dialog box from the **SignalProbe Pins** dialog box, **SignalProbe** is selected by default in the **Filter** list. Click **List** to show a set of nodes that can be probed in the **Nodes Found** list.

5. Select your source node in the **Nodes Found** list and click the ">" button. The selected node appears in the **Selected Nodes** list.

6. Click **OK**.

7. After a source is selected, the **SignalProbe enable** option is turned on. Click **Change** or **Add** to accept the changes.

Because SignalProbe pins are implemented and routed as ECOs, turning the **SignalProbe enable** option on or off is the same as selecting **Apply Selected Change** or **Restore Selected Change** in the Change Manager window. (If the Change Manager window is not visible at the bottom of your screen, from the View menu, point to **Utility Windows** and click **Change Manager**.)

For more information about the Change Manager for the Chip Planner and Resource Property Editor, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

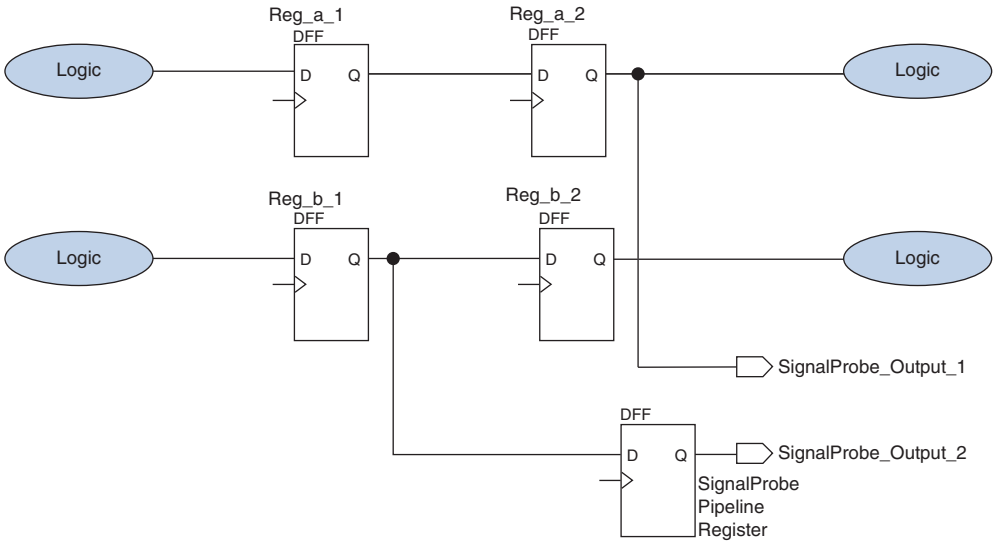
Add Registers to the Pipeline Path to SignalProbe Pin

You can specify the number of registers placed between a SignalProbe source and a SignalProbe pin to synchronize the data with a clock and to control the latency of the SignalProbe outputs. The SignalProbe feature automatically inserts the number of registers specified into the SignalProbe path.

Figure 12–2 shows a single register between the SignalProbe source `Reg_b_1` and SignalProbe `SignalProbe_Output_2` output pin added to synchronize the data between the two SignalProbe output pins.

When you add a register to a SignalProbe pin, the SignalProbe compilation attempts to place the register to best fit timing requirements. You can place SignalProbe registers near the SignalProbe source to meet f_{MAX} requirements, or you can place the register near the I/O to meet t_{CO} requirements.

Figure 12–2. Synchronizing SignalProbe Outputs with a SignalProbe Register



To pipeline an existing SignalProbe, perform the following steps:

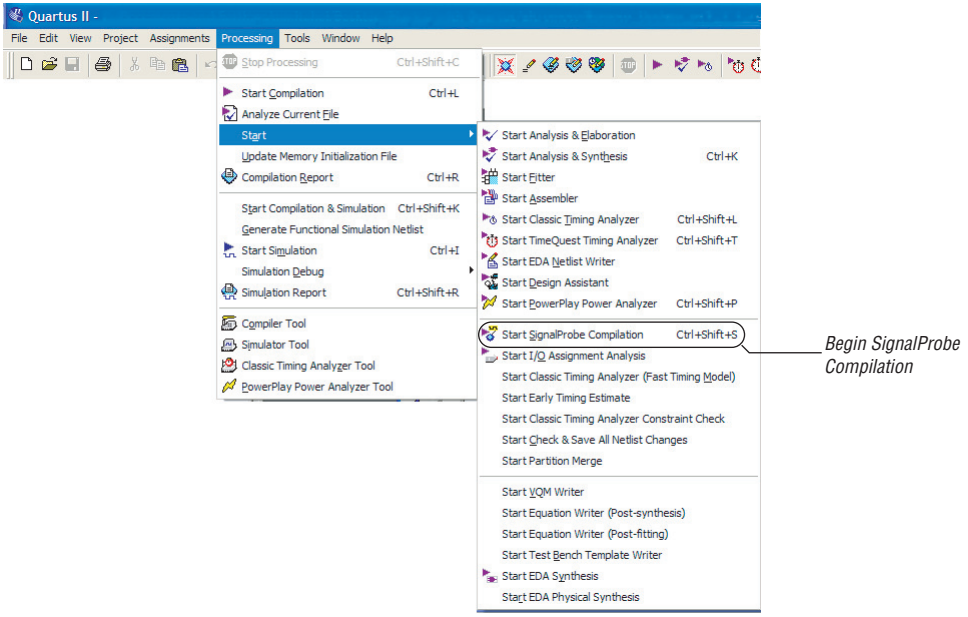
1. On the Tools menu, click **SignalProbe Pins**. The **SignalProbe Pins** dialog box appears.
2. Select a SignalProbe pin and in the **Clock** box, type the clock name used to drive your registers, or click the browse button to use the Node Finder to select your clock source.
3. In the **Registers** box, specify the number of registers you want to add in between the SignalProbe source and the SignalProbe output.
4. Click **Change.**
5. Click **OK**.

In addition to the clock input for the pipeline registers, you can also specify a reset signal pin for the pipeline registers. To specify a reset pin for the pipeline registers, use the Tcl command `make_sp` as described in "Scripting Support" on page 12–17.

Perform a SignalProbe Compilation

Perform a SignalProbe compilation to route your SignalProbe pins. On the Processing menu, point to Start and click **Start SignalProbe Compilation** (Figure 12–3). A SignalProbe compilation saves and checks all netlist changes without recompiling the other parts of the design, and completes compilation in a fraction of the time of a full compilation. The design's current placement and routing are preserved.

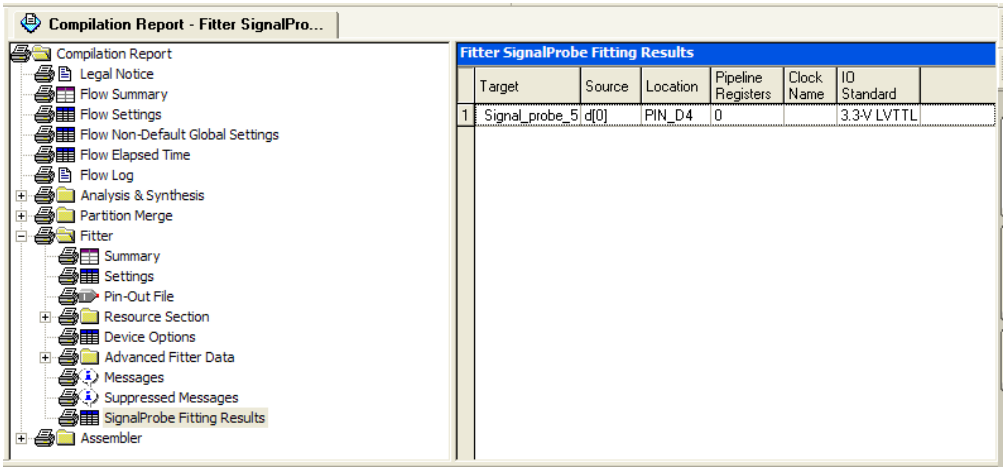
Figure 12–3. Performing the SignalProbe Compilation



Analyze the Results of the SignalProbe Compilation

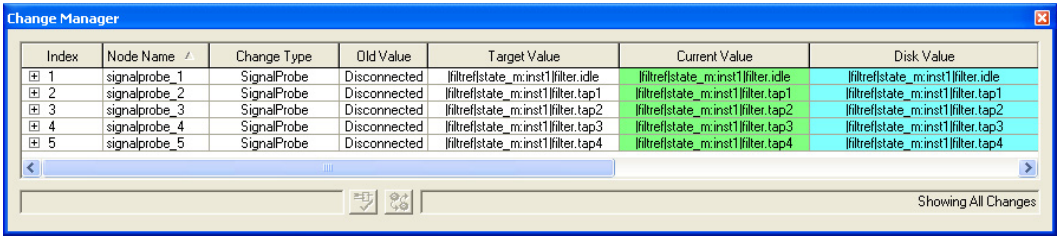
After a SignalProbe compilation, you can view the results in the compilation report file. Each SignalProbe pin is displayed in the SignalProbe Fitting Result page in the Fitter section of the Compilation Report (Figure 12–4). To view the status of each SignalProbe pin in the **SignalProbe Pins** dialog box, click **SignalProbe Pins** on the Tools menu.

Figure 12–4. SignalProbe Fitting Results Page in the Compilation Report Window



You can also view the status of each SignalProbe pin the Change Manager window (Figure 12–5). (If the Change Manager window is not visible at the bottom of your GUI, from the View menu, point to **Utility Windows** and click **Change Manager**.)

Figure 12–5. Change Manager Window with SignalProbe Pins





For more information about how to use the Change Manager, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

To view the timing results of each successfully routed SignalProbe pin, on the Processing menu, point to Start and click **Start Timing Analysis**.

Generate the Programming File

After a SignalProbe compilation, generate the new programming file containing your successfully routed SignalProbe pins. To generate a programming file, on the Processing menu, point to Start and click **Start Assembler**.

SignalProbe ECO flows

Beginning with the Quartus II software version 6.0, SignalProbe pins are implemented using the same flow as other post-compilation changes made as ECOs. The following section describes SignalProbe ECO flows with and without the Quartus II incremental compilation feature.

SignalProbe ECO Flow with Quartus II Incremental Compilation

Beginning with the Quartus II software version 6.1, the incremental compilation feature is turned on by default. The top-level design is automatically set to a design partition when the incremental compilation feature is on. A design partition during incremental compilation can have different netlist types. (Netlist types can be set to source HDL, post synthesis, or post-fit.) The netlist type indicates whether that partition should be resynthesized or refit during Quartus II incremental compilation. Incremental compilation saves you time and preserves the placement of unchanged partitions in your design if small changes must be made to some partitions late in the design cycle.



For more information about the Quartus II incremental compilation feature, refer to the *Quartus Incremental Compilation Feature for Hiearchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

The behavior of SignalProbe pins during an incremental compilation depends on the Netlist Type setting. If the top-level partition netlist type is set to post-fit, SignalProbe ECOs are retained when you recompile the design.

If some SignalProbe sources from lower-level partitions are set to a netlist type other than post-fit, then during re-compilation the Quartus II fitter uses the post-fit netlist type for those partitions as well, and a warning message appears in the message window.

All of the partitions containing SignalProbe ECOs are linked together and must use the post-fit netlist type. The same rule applies when your top-level partition is set to post-synthesis and one of the lower-level partitions' netlist type is set to post-fit. When you recompile your design, the Quartus II fitter uses the post-fit netlist for the top-level partition and SignalProbe ECOs are retained.

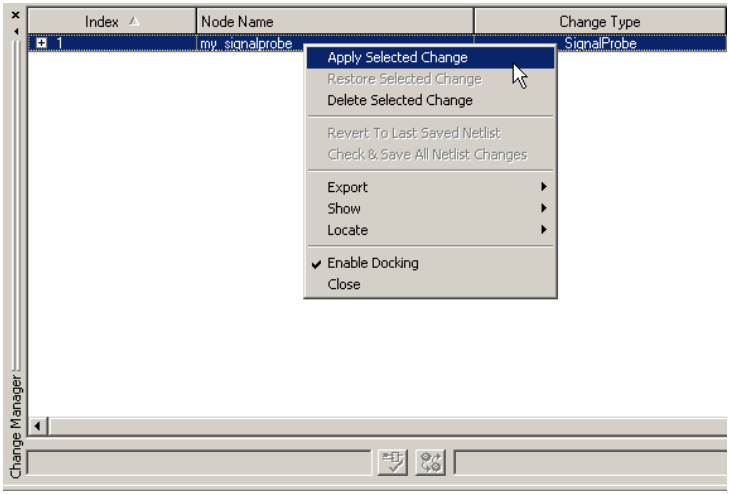
The behavior is different in the case that your top-level partition netlist type is set to post–synthesis and you have no other lower-level partitions defined, or the lower-level partition netlist types are also set to post-synthesis. If you create SignalProbe ECOs and re-compile the design, your SignalProbe ECOs are not retained and a warning message appears in the messages window. The warning indicates that ECO modifications are discarded; however, all of the ECO information is retained in the Change Manager. In this case, you can apply SignalProbe ECOs from the Change Manager and perform the **Check and Save All Netlist Changes** step as described in "SignalProbe ECO Flow without Quartus Incremental Compilation" on page 12–12.

SignalProbe ECO Flow without Quartus Incremental Compilation

If you do not use the Quartus II incremental compilation feature and you implement SignalProbe pins after the initial compilation of your design, then SignalProbe ECOs are not retained during recompilation. However, all of the SignalProbe ECOs remain in the Change Manager.

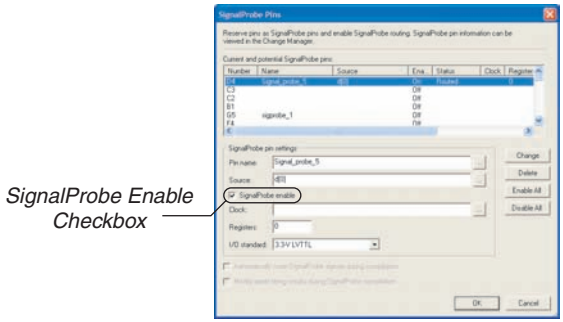
To apply a SignalProbe ECO, right-click the Change Manager and select **Apply Selected Change** (Figure 12–6). (If the Change Manager window is not visible at the bottom of your screen, from the View menu, point to **Utility Windows** and click **Change Manager**.)

Figure 12–6. Applying SignalProbe ECOs



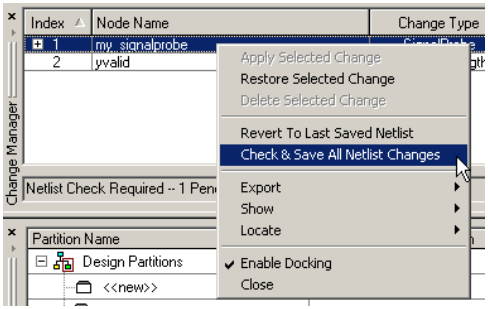
Alternately, you can use the **SignalProbe Pins** dialog box to enable the ECOs (Figure 12–7). This has the same effect as applying the SignalProbe ECOs within the Change Manager.

Figure 12–7. Enabling ECOs in the SignalProbe Pins Dialog Box



After applying the selected SignalProbe ECO, you can either click **Check and Save All Netlist Changes** from the menu within the Change Manager (Figure 12–8) or from Processing menu, point to Start and click **Start Check and Save All Netlist Changes** to perform the ECO compilation.

Figure 12–8. Check and Save All Netlist Changes



Common Questions About the SignalProbe Feature

The following are answers to common questions about the SignalProbe feature.

Why Did I Get the Following Error Message, "Error: There are No Enabled SignalProbes to Process"?

This error message is generated when a SignalProbe compilation was attempted with either no SignalProbe pins to route, or with all SignalProbe pins disabled.

This may occur if you perform a SignalProbe compilation after a full compilation. For example, when a full compilation is performed, all SignalProbe pins are disabled. You can create or re-enable your SignalProbe pins in the **SignalProbe Pins** dialog box.

How Can I Retain My SignalProbe ECOs during Re-compilation of My Design?

To retain your existing ECOs during recompilation of your design, you must use Quartus II incremental compilation. To learn more about the flow, refer to "SignalProbe ECO Flow with Quartus II Incremental Compilation" on page 12–11.

Why Did My SignalProbe Source Disappear in the Change Manager?

The SignalProbe source information for each SignalProbe is stored in the project database (**db** directory). SignalProbe pins are post-compilation changes to your netlist and are interpreted as ECOs. These changes are stored in the project **db** and if the project database is removed, the SignalProbe source information is lost and will not appear in the

SignalProbe Pins dialog box. To restore your SignalProbe pins after the design compilation step, source the **signalprobe_qsf.tcl** script located in your project directory.

You can restore your SignalProbe source information by typing the following command from a command prompt:

```
quartus_cdb -t signalprobe_qsf.tcl
```



After the compilation with Quartus II software, you must close your design project before typing the above command. Once the command finishes, you can open your design project again and the change manager shows the sources for SignalProbe pins.

What is an ECO and Where Can I Find More Information on ECO?

ECOs are late design cycle changes made to your design that do not alter functionality and timing. For more information about ECO and using the Change Manager, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

How Do I Migrate My Previous SignalProbe Assignments in the Quartus II Software Versions 5.1 and below to Versions 6.0 and Higher?

In earlier versions of the Quartus II software, SignalProbe pins were stored in the Quartus II Settings File (**.qsf**). These assignments are automatically converted into ECO changes when you open the **SignalProbe** dialog box or when you start a SignalProbe compilation in the Quartus II software versions 6.0 and higher.

For example, the SignalProbe source assignment from a Quartus II Settings File is removed and added to the Change Manager as an ECO after the **SignalProbe** dialog box is opened, or when you perform a SignalProbe compilation.

Example 12–1. SignalProbe Assignments in the Quartus II Settings File

```
set_location_assignment PIN_C22 -to my_signalprobe_pin
set_instance_assignment -name RESERVE_PIN "AS SIGNALPROBE OUTPUT" -to my_signalprobe_pin
set_instance_assignment -name IO_STANDARD LVTTL -to my_signalprobe_pin
set_instance_assignment -name SIGNALPROBE_ENABLE ON -to my_signalprobe_pin
set_instance_assignment -name SIGNALPROBE_SOURCE inst5[0] -to my_signalprobe_pi
```

Example 12–2. SignalProbe Assignments in the Quartus II Settings File after Opening the SignalProbe Pins Dialog Box

```
set_location_assignment PIN_C22 -to my_signalprobe_pin
set_instance_assignment -name RESERVE_PIN "AS SIGNALPROBE OUTPUT" -to my_signalprobe_pin
set_instance_assignment -name IO_STANDARD LVTTL -to my_signalprobe_pin
set_instance_assignment -name SIGNALPROBE_ENABLE ON -to my_signalprobe_pin
```

What are all the Changes for the SignalProbe Feature between the Quartus II Software Version 5.1 and Earlier, and Version 6.0 and Later?

The following list of changes affect users of the SignalProbe feature in the Quartus II software versions 5.1 and below with Stratix series, Cyclone series, and MAX II device families.

- In Quartus II software versions 5.1 and earlier, the **SignalProbe Pins** dialog box was accessed on the Assignments menu. To access it with the Quartus II software version 6.0 and later, on the Tools menu, click **SignalProbe Pins**.
- A full compilation is required before making SignalProbe connections. However, you can still reserve pins before compilation for later use by SignalProbe. You can reserve pins by creating a SignalProbe in the **SignalProbe** dialog box without specifying a source. This is the same behavior as in the Quartus II software version 5.1.
- To route the SignalProbe pins, you must perform a SignalProbe compilation after a full compilation. The **Automatically route SignalProbe signals during compilations** and **Modify latest fitting results during SignalProbe compilation** options are no longer supported.
- After subsequent compiles, full or incremental, existing SignalProbe pins are disabled and are not present in the post-compilation netlist. To add them back, enable the SignalProbe pins and perform a SignalProbe compilation.
- SignalProbe pins are not controlled via assignments in the Quartus II Settings File because they are now ECOs. Existing Quartus II Settings Files automatically convert to ECOs when a SignalProbe compilation is performed or when the **SignalProbe** dialog box is opened.
- The Tcl interface for creating SignalProbe pins has improved and is a part of the Chip Planner package `::quartus::chip_editor`. Refer to "Scripting Support" on page 12–17.
- Previously, the `quartus_fit --signalprobe` command was used to perform a SignalProbe compilation. This is not supported in the Quartus II software version 6.0 and later, and is replaced by the improved Tcl interface and the `check_netlist_and_save` Tcl command.

- The SignalProbe timing report generated after a successful SignalProbe compilation is not available in the Quartus II software version 6.0 and later. You can view the timing results of your SignalProbe pins in the SignalProbe Fitting Results, under the Fitter report, or in the t_{CO} results page of the Timing report.
- You can not make SignalProbe pins in the Assignment Editor. Use the **SignalProbe Pins** dialog box to make and edit your SignalProbe pins.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II command-line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Scripting Reference Manual* includes the same information in PDF form.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Reference Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Make a SignalProbe Pin

You can make a SignalProbe pin by typing the following command:

```
make_sp [-h | -help] [-long_help] [-clk <clk>] [-io_std <io_std>] \
-loc <loc> -pin_name <pin name> [-regs <regs>] [-reset <reset>] \
-src_name <source name> ↵
```

Delete a SignalProbe Pin

You can delete a SignalProbe pin by typing the following command:

```
delete_sp [-h | -help] [-long_help] -pin_name <pin name> ↵
```

Enable a SignalProbe Pin

You can enable a SignalProbe pin by typing the following command:

```
enable_sp [-h | -help] [-long_help] -pin_name <pin name> ↵
```

Disable a SignalProbe Pin

You can disable a SignalProbe pin by typing the following command:

```
disable_sp [-h | -help] [-long_help] -pin_name <pin name> ↵
```

Perform a SignalProbe Compilation

You can perform a SignalProbe compilation by typing the following command:

```
check_netlist_and_save ↵
```

Migrating Previous SignalProbe Pins to the Quartus II Software Versions 6.0 and Later

You can migrate previous SignalProbe pins to the Quartus II software versions 6.0 and later by typing the following command:

```
convert_signal_probes ↵
```

Script Example

Example 12–3 is a script that creates a SignalProbe pin called `sp1` and connecting it to source node `reg1` in a project that was already compiled.

Example 12–3. Creating a SignalProbe Pin Called sp1

```
Package require ::quartus::chip_editor
Project_open project
Read_netlist
Make_sp -pin_name sp1 -src_name reg1
Check_netlist_and_save
Project_close
```

Using SignalProbe with the APEX Device Family

APEX devices do not support post-fit netlist changes made as ECOs. You can use SignalProbe compilation to route internal signals to output pins incrementally. The SignalProbe incremental routing feature does not affect design behavior.

To use the SignalProbe feature, follow these steps:

1. Reserve SignalProbe pins. For more information, refer to "Reserve the SignalProbe Pins" on page 12–4.
2. Assign a SignalProbe source to each SignalProbe pin.
3. Perform a SignalProbe compilation.
4. Analyze the results of a SignalProbe compilation.

Adding SignalProbe Sources

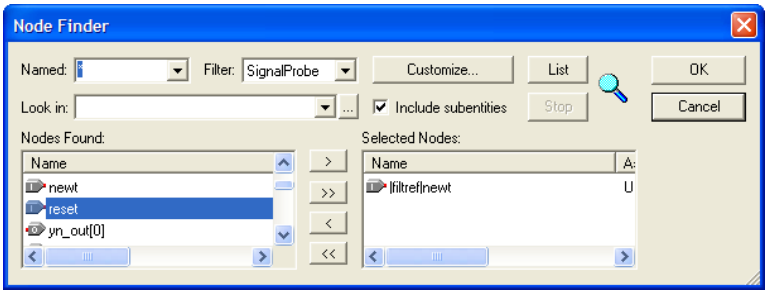
A SignalProbe source is a signal in the post-compilation design database with a possible route to an output pin. You can assign a SignalProbe source to a SignalProbe pin, or an unused output pin by performing the following steps:

1. On the Tools menu, click **SignalProbe Pins**. The **SignalProbe Pins** dialog box appears.
2. In the **Current and potential SignalProbe pins** list, select the SignalProbe pin to which you want to add a SignalProbe source.
3. Click **Browse** and select a SignalProbe source.
4. Click **OK**.

   The **Node Finder** dialog box displays with the SignalProbe filter selected (Figure 12–9). Click **List** to view all of the available SignalProbe sources. If you cannot find a specific node with the

SignalProbe filter, then the node either has either been removed by the Quartus II software during optimization, or placed in the device where there are no possible routes to a pin.

Figure 12–9. Available SignalProbe Sources in the Node Finder



5. In the **Assign SignalProbe Pins** dialog box, click **Add** if a source has not been assigned to the SignalProbe pin.

   or

   Click **Change** for a SignalProbe pin that has a source already assigned.

When the source of the SignalProbe pin is added or changed, the SignalProbe pin is automatically enabled. To disable a SignalProbe pin, turn off **SignalProbe enable**.

6. Click **OK.**

Performing a SignalProbe Compilation

You can start a SignalProbe compilation manually or automatically after a full compilation. A SignalProbe compilation includes the following:

- Validates SignalProbe pins.
- Validates your specified SignalProbe sources.
- If applicable, adds registers into SignalProbe paths.
- Attempts to route from SignalProbe sources through registers to SignalProbe pins.

To run the SignalProbe compilation automatically after a full compilation, on the Tools menu, click **SignalProbe Pins**. In the **SignalProbe Pins** dialog box, turn on **Automatically route SignalProbe signals during compilation**.

To run a SignalProbe compilation manually after a full compilation, on the Processing menu, point to Start and click **Start SignalProbe Compilation**.



You must run the Fitter before a SignalProbe compilation. The Fitter generates a list of all internal nodes that can be used as SignalProbe sources.

You can enable and disable each SignalProbe pin by turning the **SignalProbe enable** option on and off in the **SignalProbe Pins** dialog box.

Running SignalProbe with Smart Compilation

Optimally, you can run a smart compilation, which reduces compilation time by running only necessary modules during compilation. However, a full compilation is required if any design files, Analysis and Synthesis settings, or Fitter settings have changed.

To turn on smart compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings and** turn on **Use Smart compilation**.

If you run a SignalProbe compilation with smart compilation turned on, and there are changes to a design file or settings related to the Analysis and Synthesis or Fitter modules, the following message is displayed:

```
Error: Can't perform SignalProbe compilation because design
requires a full compilation.
```



You should turn smart compilation on, which allows you to work with the latest settings and design files.

Understanding the Results of a SignalProbe Compilation

After a SignalProbe compilation, the results appear in two sections of the compilation report file. The fitting results and status (Table 12–2) of each SignalProbe pin is displayed in the SignalProbe Fitting Result page in the Fitter section of the compilation report (Figure 12–10).

The timing results of each successfully routed SignalProbe pin is displayed in the **SignalProbe source to output delays** page in the Timing Analysis section of the compilation report (Figure 12–11).

After a SignalProbe compilation, the processing page of the Messages window also provides the results of each SignalProbe pin and displays slack information for each successfully routed SignalProbe pin.

Table 12–2. Status Values

Status	Description
Routed	Connected and routed successfully
Not Routed	Not enabled
Failed to Route	Failed routing during last SignalProbe compilation
Need to Compile	Assignment changed since last SignalProbe compilation

Figure 12–10. SignalProbe Fitting Results Page in the Compilation Report Window

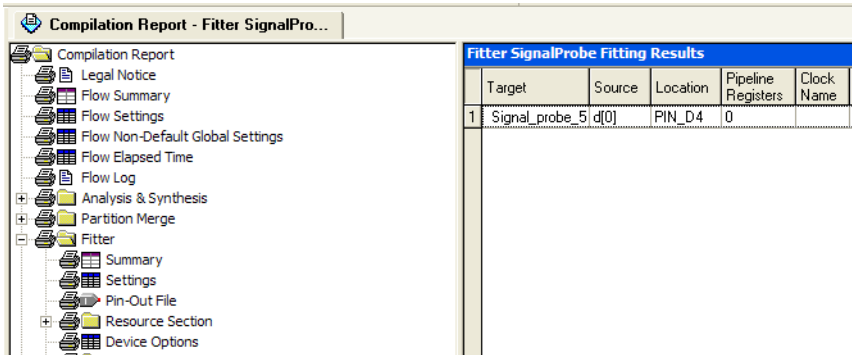
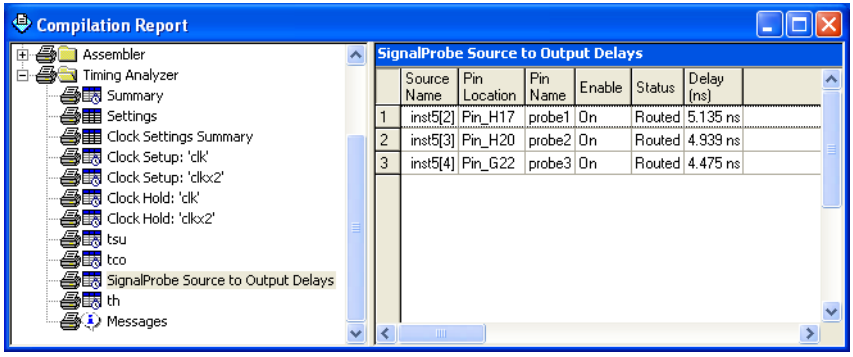


Figure 12–11. SignalProbe Source to Output Delays Page in the Compilation Report Window



Analyzing SignalProbe Routing Failures

The SignalProbe can begin compilation; however, one of the following reasons can prevent complete compilation:

- **Route unavailable**—the SignalProbe compilation failed to find a route from the SignalProbe source to the SignalProbe pin because of routing congestion
- **Invalid or nonexistent SignalProbe source**—you entered a SignalProbe source that does not exist or is invalid
- **Unusable output pin**—the output pin selected is found to be unusable

Routing failures can occur if the SignalProbe pin's I/O standard conflicts with other I/O standards in the same I/O bank.

If routing congestion prevents a successful SignalProbe compilation, you can allow the compiler to modify the routing to the specified SignalProbe source. On the Tools menu, click **SignalProbe Pins** and turn on **Modify latest fitting results during SignalProbe compilation**. This setting allows the Fitter to modify existing routing channels used by your design.



Turning on **Modify latest fitting results during SignalProbe compilation** can change the performance of your design.

SignalProbe Scripting Support for APEX Devices

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Scripting Reference Manual* includes the same information in PDF form.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Reserving SignalProbe Pins

Use the following Tcl commands to reserve a SignalProbe pin.

```
set_location_assignment <location> -to <SignalProbe pin name>

set_instance_assignment -name RESERVE_PIN \
"AS SIGNALPROBE OUTPUT" -to <SignalProbe pin name> ↵
```

Valid locations are pin location names, such as `Pin_A3`.

For more information about reserving SignalProbe pins, refer to "Reserve the SignalProbe Pins" on page 12–4.

Adding SignalProbe Sources

Use the following Tcl commands to add SignalProbe sources. For more information about adding SignalProbe sources, refer to "Adding SignalProbe Sources" on page 12–19. The following command assigns the node name to a SignalProbe pin:

```
set_instance_assignment -name SIGNALPROBE_SOURCE \
<node name> -to <SignalProbe pin name> ↵
```

The next command turns on the SignalProbe routing. You can turn off individual SignalProbe pins by specifying `OFF` instead of `ON` with the following command:

```
set_instance_assignment -name SIGNALPROBE_ENABLE ON \
-to <SignalProbe pin name> ↵
```

Assigning I/O Standards

Use the following Tcl command to assign an I/O standard to a pin:

```
set_instance_assignment -name IO_STANDARD \
<I/O standard> -to <SignalProbe pin name> ↵
```



For a list of valid I/O standards, refer to the I/O Standards general description in the Quartus II Help.

Adding Registers for Pipelining

Use the following Tcl commands to add registers for pipelining:

```
set_instance_assignment -name SIGNALPROBE_CLOCK \
<clock name> -to <SignalProbe pin name>
```

```
set_instance_assignment \
-name SIGNALPROBE_NUM_REGISTERS <number of registers> \
-to <SignalProbe pin name> ↵
```

Run SignalProbe Automatically

Use the following Tcl command to run SignalProbe automatically after a full compile.

```
set_global_assignment -name \
SIGNALPROBE_DURING_NORMAL_COMPILATION ON
```

For more information about running SignalProbe automatically, refer to "Performing a SignalProbe Compilation" on page 12–20.

Run SignalProbe Manually

You can run SignalProbe manually with a Tcl command or the `quartus_fit` command at a command prompt.

```
execute_flow -signalprobe ↵
```

The `execute_flow` command is in the flow package. At a command prompt, type the following command:

```
quartus_fit <project name> --signalprobe ↵
```

For more information about running SignalProbe manually, refer to "Performing a SignalProbe Compilation" on page 12–20.

Enable or Disable All SignalProbe Routing

Use the Tcl command in Example 12–4 to turn on or turn off SignalProbe routing. In the `set_instance_assignment` command, specify `ON` to turn on SignalProbe routing or `OFF` to turn off SignalProbe routing.

Example 12–4. Turning SignalProbe On or Off with Tcl

```
set spe [get_all_assignments -name SIGNALPROBE_ENABLE] \
foreach_in_collection asgn $spe {
    set signalprobe_pin_name [lindex $asgn 2]
    set_instance_assignment -name SIGNALPROBE_ENABLE -to \
$signalprobe_pin_name <ON|OFF> } ↵
```

For more information about enabling or disabling SignalProbe routing, refer to page 12–20.

Running SignalProbe with Smart Compilation

Use the following Tcl command to turn on **Smart Compilation**:

```
set_global_assignment -name SMART_RECOMPILE ON ↵
```

For more information, refer to "Running SignalProbe with Smart Compilation" on page 12–21.

Allow SignalProbe to Modify Fitting Results

Use the following Tcl command to turn on **Modify latest fitting results**.

```
set_global_assignment -name \
SIGNALPROBE_ALLOW_OVERUSE ON ↵
```

For more information, refer to "Analyzing SignalProbe Routing Failures" on page 12–23.

Conclusion

Using the SignalProbe feature can significantly reduce the time required compared to a full recompilation. You can use the SignalProbe feature to get quick access to internal design signals to perform system-level debugging.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 12–3 shows the revision history for this chapter.

Table 12–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 12–26.	—
May 2007 v7.1.0	Added Referenced Documents, minor updates to address ADoQS issues.	—
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Updated for the Quartus II software version 6.1.0: • New section (SignalProbe ECO flows) added to explain how SignalProbe pins' ECOs are affected during Quartus II Incremental Compilation. • QandA added to answer: How Can I Retain My SignalProbe ECOs during Re-compilation of My Design.	Quartus II software version 6.1.0 added more ECO features; the chapter updated to reflect this change.
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.0: • Documented new SignalTap features.	—
December 2005 v5.1.1	Added SMART_RECOMPILE assignment.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
May 2005 v5.0.0	• Minor updates for Quartus II software 5.0	—
December 2004 v2.1	• Chapter 9 was formerly Chapter 8. • Updates to tables and figures. • New functionality for Quartus II software 4.2.	—
June 2004 v2.0	• Updates to tables, figures. • New functionality for Quartus II software 4.1.	—
February 2004 v1.0	Initial release.	—

13. Design Debugging Using the SignalTap II Embedded Logic Analyzer

QII53009-7.2.0

Introduction

The phenomenal growth in design size and complexity continues to make design verification a critical bottleneck for current FPGA systems. Limited access to internal signals, complex FPGA packages, and PCB electrical noise all contribute to making design debugging the most challenging process of the design cycle. More than 50% of the design cycle time can be spent on debugging and verifying the design. To help with the process of design debugging, Altera® provides a solution that enables a designer to examine the behavior of internal signals, without using extra I/O pins, while the design is running at full speed on an FPGA device.

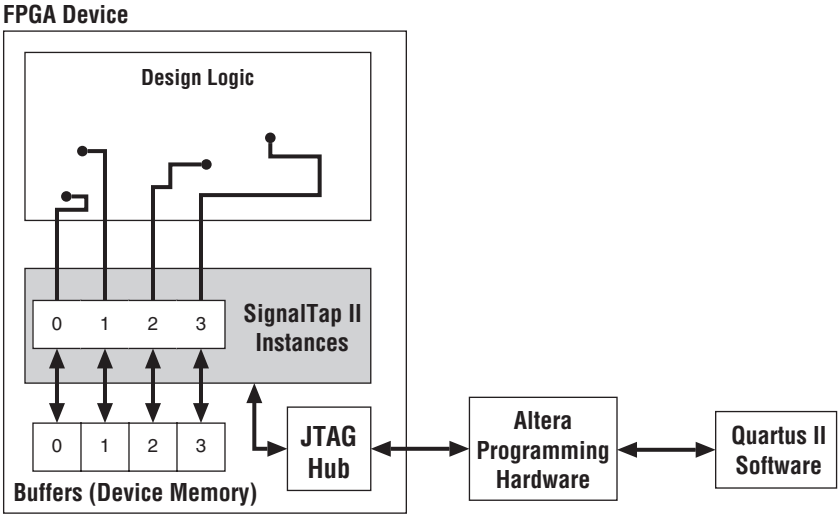
The SignalTap® II Embedded Logic Analyzer is scalable, easy to use, and is included with the Quartus® II software subscription. This logic analyzer helps debug an FPGA design by probing the state of the internal signals in the design without the use of external equipment. Defining custom trigger-condition logic provides greater accuracy and improves the ability to isolate problems. The SignalTap II Embedded Logic Analyzer does not require external probes, or changes to the design files to capture the state of the internal nodes or I/O pins in the design. All captured signal data is conveniently stored in device memory until the designer is ready to read and analyze the data.

The topics in this chapter include:

- "On-Chip Debugging Tool Comparison" on page 13–5
- "Design Flow Using the SignalTap II Logic Analyzer" on page 13–7
- "SignalTap II Logic Analyzer Task Flow" on page 13–8
- "Add the SignalTap II Logic Analyzer to Your Design" on page 13–10
- "Configure the SignalTap II Logic Analyzer" on page 13–18
- "Define Triggers" on page 13–30
- "Program the Target Device or Devices" on page 13–57
- "Run the SignalTap II Logic Analyzer" on page 13–59
- "View, Analyze, and Use Captured Data" on page 13–63
- "Other Features" on page 13–67
- "SignalTap II Scripting Support" on page 13–72
- "Design Example: Using SignalTap II Logic Analyzers in SOPC Builder Systems" on page 13–77
- "Custom Triggering Flow Application Examples" on page 13–77

The SignalTap II Embedded Logic Analyzer is a next-generation, system-level debugging tool that captures and displays real-time signal behavior in a system-on-a-programmable-chip (SOPC) or any FPGA design. The SignalTap II Embedded Logic Analyzer supports the highest number of channels, largest sample depth, and fastest clock speeds of any embedded logic analyzer in the programmable logic market. Figure 13–1 shows a block diagram of the components that make up the SignalTap II Embedded Logic Analyzer.

Figure 13–1. SignalTap II Logic Analyzer Block Diagram *Note (1)*



Note to Figure 13–1:

(1) This diagram assumes that the SignalTap II Logic Analyzer was compiled with the design as a separate design partition using the Quartus II Incremental Compilation feature. This is the default setting for new projects in the Quartus II software. If incremental compilation is disabled or not used, the SignalTap II logic is integrated with the design. For information about the use of incremental compilation with SignalTap II, refer to "Faster Compilations with Quartus II Incremental Compilation" on page 13–51.

This chapter is intended for any designer who wants to debug their FPGA design during normal device operation without the need for external lab equipment. Because the SignalTap II Embedded Logic Analyzer is similar to traditional external logic analyzers, familiarity with external logic analyzer operations is helpful but not necessary. To take advantage of faster compile times when making changes to the SignalTap II Logic Analyzer, knowledge of the Quartus II Incremental Compilation feature is helpful.

For information about using the Quartus II Incremental Compilation feature, refer to the *Incremental Compilation for Hierarchical and Team-Based Design* chapter in the *Quartus II Handbook*.

Hardware and Software Requirements

The following components are required to perform logic analysis with the SignalTap II Embedded Logic Analyzer:

- Quartus II design software
  or
  Quartus II Web Edition (with TalkBack feature enabled)
  or
  SignalTap II Logic Analyzer standalone software

- Download/Upload Cable
- Altera development kit or user design board with JTAG connection to device under test

Captured data is stored in the device's memory blocks and transferred to the Quartus II software waveform display with a JTAG communication cable, such as EthernetBlaster or USB-Blaster™. Table 13–1 summarizes some of the features and benefits of the SignalTap II Embedded Logic Analyzer.

Table 13–1. SignalTap II Features and Benefits (Part 1 of 2)

Feature	Benefit
Multiple logic analyzers in a single device	Captures data from multiple clock domains in a design at the same time
Multiple logic analyzers in multiple devices in a single JTAG chain	Simultaneously captures data from multiple devices in a JTAG chain
Plug-In Support	Easily specifies nodes, triggers, and signal mnemonics for IP, such as the Nios® II embedded processor
Up to 10 basic or advanced trigger conditions for each analyzer instance	Enables more complex data capture commands to be sent to the logic analyzer, providing greater accuracy and problem isolation
Power-Up Trigger	Captures signal data for triggers that occur after device programming but before manually starting the logic analyzer
State-Based Triggering Flow	Enables you to organize your triggering conditions to precisely define what your embedded logic analyzer will capture
Incremental Compilation	Modifies the SignalTap II Logic Analyzer monitored signals and triggers without performing a full compilation, saving time
Flexible buffer acquisition modes	Allows more accurate data collection by setting each trigger to sample at different ranges relative to the triggering event, in circular or segmented modes

Table 13–1. SignalTap II Features and Benefits (Part 2 of 2)

Feature	Benefit
MATLAB integration with included MEX function	Acquires the SignalTap II Logic Analyzer captured data into a MATLAB integer matrix
Up to 1,024 channels in each device	Samples many signals and wide bus structures
Up to 128K samples in each device	Captures a large sample set for each channel
Fast clock frequencies	Collects sample data at up to 270 MHz
Resource usage estimator	Provides estimate of logic and memory device resources used by SignalTap II Embedded Logic Analyzer configurations
No additional cost	The SignalTap II Logic Analyzer is included with a Quartus II subscription and with the Quartus II Web Edition (with TalkBack enabled)



For a list of supported device families, refer to the Quartus II Help.

On-Chip Debugging Tool Comparison

The Quartus II software provides a number of different ways to help debug your FPGA design after programming the device. The SignalTap II Logic Analyzer, SignalProbe, and the Logic Analyzer Interface (LAI) share some similar features, but each has its own advantages. In some debugging situations, it can be difficult to decide which tool is best to use or whether multiple tools are required. Table 13–2 compares common debugging features between these tools and provides suggestions about which is the best tool to use for a given feature.

Table 13–2. Suggested On-Chip Debugging Tools for Common Debugging Features Note (1) (Part 1 of 2)

Feature	SignalProbe	Logic Analyzer Interface (LAI)	SignalTap II Embedded Analyzer	Description
Large Sample Depth	N/A	✓	—	An external logic analyzer used with the LAI has a bigger buffer to store more captured data than the SignalTap II Logic Analyzer. No data is captured or stored with SignalProbe.
Ease in Debugging Timing Issue	N/A	✓	—	An external logic analyzer used with the LAI provides you with access to timing mode, enabling you to debug combined streams of data.
Minimal Effect on Logic Design	✓	✓ *(2)*	✓ *(2)*	The LAI adds minimal logic to a design, requiring fewer device resources. The SignalTap II Logic Analyzer has little effect on the design when it is set as a separate design partition using incremental compilation. SignalProbe incrementally routes nodes to pins, not affecting the design at all.
Short Compile and Recompile Time	✓	✓ *(2)*	✓ *(2)*	SignalProbe attaches incrementally routed signals to previously reserved pins, requiring very little recompilation time to make changes to source signal selections. The SignalTap II Logic Analyzer and the LAI can take advantage of incremental compilation to refit their own design partitions to decrease recompilation time.
Triggering Capability	N/A	✓	✓	The SignalTap II Logic Analyzer offers triggering capabilities that are comparable to commercial logic analyzers.
I/O Usage	—	—	✓	No additional output pins are required with the SignalTap II Logic Analyzer. Both the LAI and SignalProbe require I/O pin assignments.

Table 13–2. Suggested On-Chip Debugging Tools for Common Debugging Features Note (1) (Part 2 of 2)

Feature	SignalProbe	Logic Analyzer Interface (LAI)	SignalTap II Embedded Analyzer	Description
Acquisition Speed	N/A	—	✓	The SignalTap II Logic Analyzer can acquire data at speeds of over 200 MHz. The same acquisition speeds are obtainable with an external logic analyzer used with the LAI, but signal integrity issues may limit this.
No JTAG Connection Required	✓	—	—	An FPGA design with the SignalTap II Logic Analyzer or the LAI requires an active JTAG connection to a host running the Quartus II software. SignalProbe does not require a host for debugging purposes.
External Equipment	—	—	✓	The SignalTap II Logic Analyzer logic is completely internal to the programmed FPGA device. No extra equipment is required other than a JTAG connection from a host running the Quartus II software or the stand-alone SignalTap II software. SignalProbe and the LAI require the use of external debugging equipment, such as multimeters, oscilloscopes, or logic analyzers.

Notes to Table 13–2:
(1) ✓ indicates the suggested best tool for the feature. — indicates that while the tool is available for that feature, that tool may not give the best results. N/A indicates that the feature is not applicable for the selected tool.
(2) When used with incremental compilation.

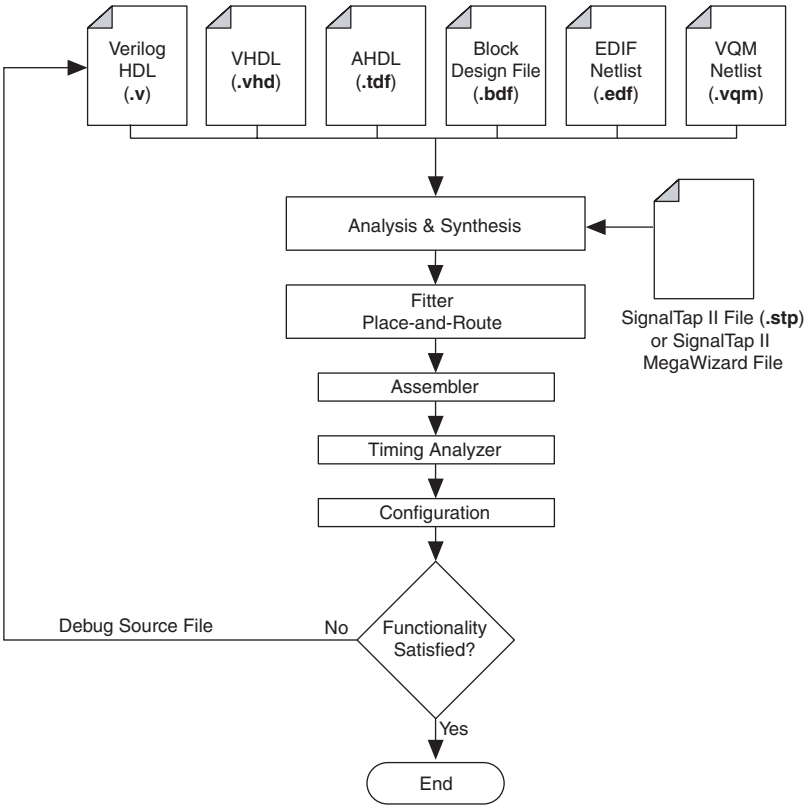
If you have signals that you want to monitor with external equipment without adding the logic resources required by the SignalTap II Logic Analyzer, consider the use of these other tools available in the Quartus II software. Signals can be quickly routed out to reserved I/O pins as part of an ECO change using SignalProbe, while multiplexed banks of many signals can be made visible on only a few pins with the use of the LAI.

For information about the use of these tools, refer to the *Quick Design Debugging Using SignalProbe* and *In-System Debugging Using External Logic Analyzers* chapters in volume 3 of the *Quartus II Handbook*.

Design Flow Using the SignalTap II Logic Analyzer

Figure 13–2 shows a typical overall FPGA design flow for using the SignalTap II Logic Analyzer in your design. A SignalTap II file (**.stp**) is added to and enabled in your project, or a SignalTap II HDL function, created with the MegaWizard® Plug-In Manager, is instantiated in your design. The diagram shows the flow of operations from initially adding the SignalTap II Logic Analyzer to your design to the final device configuration, testing, and debugging.

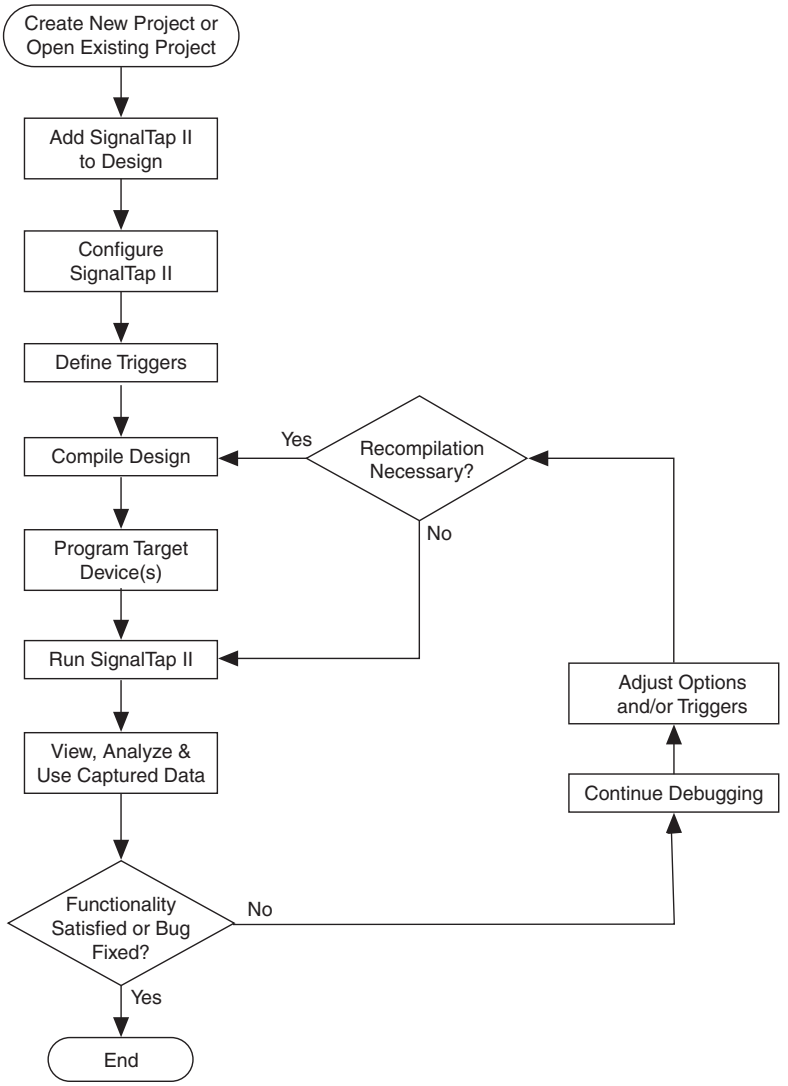
Figure 13–2. SignalTap II FPGA Design and Debugging Flow



SignalTap II Logic Analyzer Task Flow

To use the SignalTap II Logic Analyzer to debug your design, you perform a number of tasks to add, configure, and run the logic analyzer. Figure 13–3 shows a typical flow of the tasks you complete to debug your design. Refer to the appropriate section of this chapter for more information about each of these tasks.

Figure 13–3. SignalTap II Logic Analyzer Task Flow



Add the SignalTap II Logic Analyzer to Your Design

Create a SignalTap II file or create a parameterized HDL instance representation of the logic analyzer using the MegaWizard Plug-In Manager. If you want to monitor multiple clock domains simultaneously, you can add additional instances of the logic analyzer to your design, limited only by the available resources in your device.

Configure the SignalTap II Logic Analyzer

Once the SignalTap II Logic Analyzer is added to your design, you configure it to monitor the signals you want. You can manually add signals or use a plug-in, such as the Nios II plug-in, to quickly add entire sets of associated signals for a particular IP. You can also specify settings for the data capture buffer, such as its size, the method in which data is captured and stored, and the device memory type to use for the buffer in devices that support memory type selection.

Define Triggers

The SignalTap II Logic Analyzer continuously captures data while it is running. To capture and store specific signal data, you set up triggers that tell the logic analyzer under what conditions to stop capturing data. The SignalTap II Logic Analyzer lets you define Runtime Triggers that range from very simple, such as the rising edge of a single signal, to very complex, involving groups of signals, extra logic, and multiple conditions. Power-Up Triggers give you the ability to capture data from trigger events occurring immediately after the device enters user-mode after configuration.

Compile the Design

With the SignalTap II file configured and triggers defined, you compile your project as usual to include the logic analyzer in your design. Since you may need to frequently change monitored signal nodes or adjust trigger settings during debugging, it is recommended that you use the incremental compilation feature built into the SignalTap II Logic Analyzer, along with Quartus II incremental compilation, to reduce recompile times.

Program the Target Device or Devices

When you are debugging a design with the SignalTap II Logic Analyzer, you can program a target device directly from the SignalTap II file without using the Quartus II Programmer. You can also program multiple devices with different designs and simultaneously debug them.

Run the SignalTap II Logic Analyzer

In normal device operation, you control the logic analyzer through the JTAG connection, specifying when to start looking for your trigger conditions to begin capturing data. With Runtime or Power-Up Triggers, you read and transfer the captured data from the on-chip buffer to the SignalTap II file for analysis.

View, Analyze, and Use Captured Data

Once you have captured data and read it into the SignalTap II file, it is available for analysis and use in the debugging process. Either manually or with a plug-in, you can set up mnemonic tables to make it easier to read and interpret the captured signal data. To speed up debugging, use the Locate feature in the SignalTap II node list to find the locations of problem nodes in other tools in the Quartus II software. Save the captured data for later analysis, or convert it to other formats for sharing and further study.

Add the SignalTap II Logic Analyzer to Your Design

Because the SignalTap II Logic Analyzer is implemented in logic on your target device, it must be added to your FPGA design as another part of the design itself. There are two ways to generate the SignalTap II Logic Analyzer and add it to your design for debugging:

- Create a SignalTap II file (**.stp**) and use the SignalTap II Editor to configure the details of the logic analyzer
- Create and configure the SignalTap II file with the MegaWizard Plug-In Manager and instantiate it in your design

Creating and Enabling a SignalTap II File

To create an embedded logic analyzer, you can use an existing SignalTap II file or create a new file. Once a file is created or selected, it must be enabled in the project where it is used.

Creating a SignalTap II File

The SignalTap II file contains the SignalTap II Logic Analyzer settings and the captured data for viewing and analysis. To create a new SignalTap II file, perform the following steps:

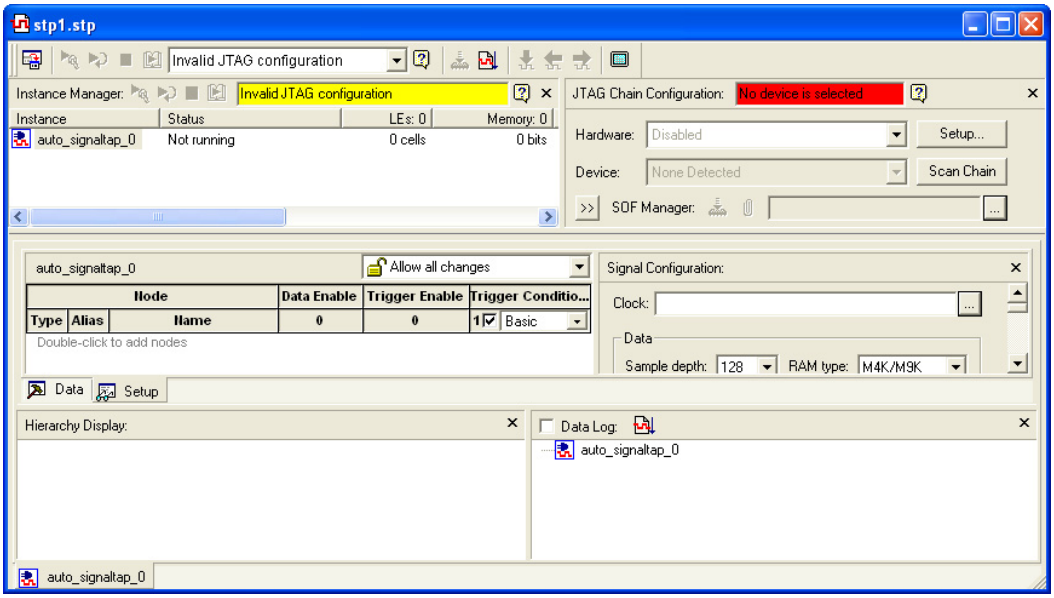
1. On the File menu, click **New**.
2. In the **New** dialog box, click the **Other Files** tab, and select **SignalTap II Logic Analyzer File**.

3. Click **OK**.

To open an existing SignalTap II file already associated with your project, on the Tools menu, click **SignalTap II Logic Analyzer**. You can also use this method to create a new SignalTap II file if no SignalTap II file exists for the current project.

To open an existing file, on the File menu, click **Open** and select a SignalTap II file (Figure 13–4).

Figure 13–4. SignalTap II Editor



Enabling and Disabling a SignalTap II File for the Current Project

Whenever you save a new SignalTap II file, the Quartus II software asks you if you want to enable the file for the current project. However, you can add this file manually, change the selected SignalTap II file, or completely disable the logic analyzer by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.

2. In the **Category** list, select **SignalTap II Logic Analyzer.** The **SignalTap II Logic Analyzer** page appears.

3. Turn on **Enable SignalTap II Logic Analyzer.** Turn off this option to disable the logic analyzer, completely removing it from your design.

4. In the **SignalTap II File name** box, type the name of the SignalTap II file you want to include with your design, or browse to and select a file name.

5. Click **OK**.

Using the MegaWizard Plug-In Manager to Create Your Embedded Logic Analyzer

Alternatively, you can create a SignalTap II Logic Analyzer instance by using the MegaWizard Plug-In Manager. The MegaWizard Plug-In Manager generates an HDL file that you instantiate in your design. You can also use a hybrid approach in which you instantiate the MegaWizard Plug-In Manager file in your HDL, and then use the method described in "Creating and Enabling a SignalTap II File" on page 13–10.

Creating an HDL Representation Using the MegaWizard Plug-In Manager

The Quartus II software allows you to easily create your SignalTap II Logic Analyzer using the MegaWizard Plug-In Manager. To implement the SignalTap II megafunction, perform the following steps:

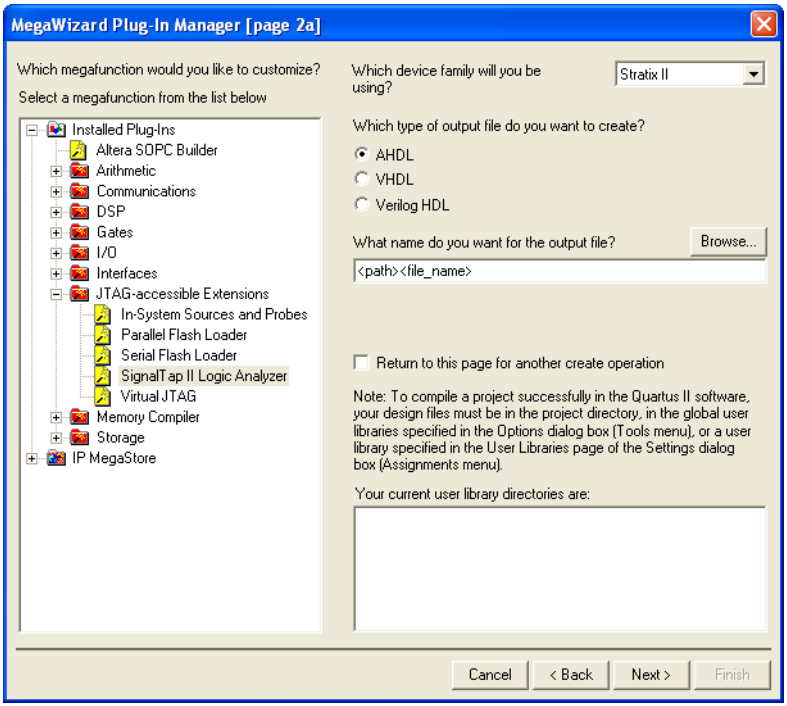
1. On the Tools menu, click **MegaWizard Plug-In Manager**. Page 1 of the **MegaWizard Plug-In Manager** dialog box appears.

2. Select **Create a new custom megafunction variation**.

3. Click **Next**.

4. In the **Installed Plug-Ins** list, expand the **JTAG-accessible Extensions** folder, and select **SignalTap II Logic Analyzer**. Select an output file type and enter the desired name of the SignalTap II megafunction. You can choose AHDL (**.tdf**), VHDL (**.vhd**), or Verilog HDL (**.v**) as the output file type (Figure 13–5).

Figure 13–5. Creating the SignalTap II Logic Analyzer in the MegaWizard Plug-In Manager

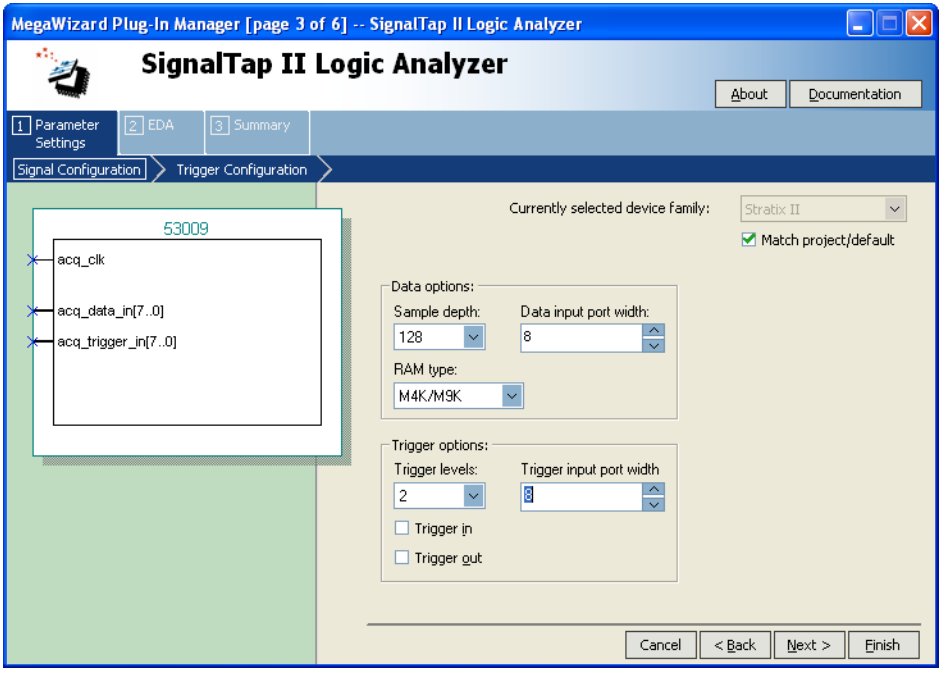


5. Click **Next**.

6. Configure the analyzer by specifying the **Sample depth**, **RAM Type**, **Data input port width**, **Trigger levels**, **Trigger input port width**, and whether to enable an external **Trigger in** or **Trigger out** (Figure 13–6).

   For information about these settings, refer to "Configure the SignalTap II Logic Analyzer" on page 13–18 and "Define Triggers" on page 13–30.

Figure 13–6. Select Logic Analyzer Parameters



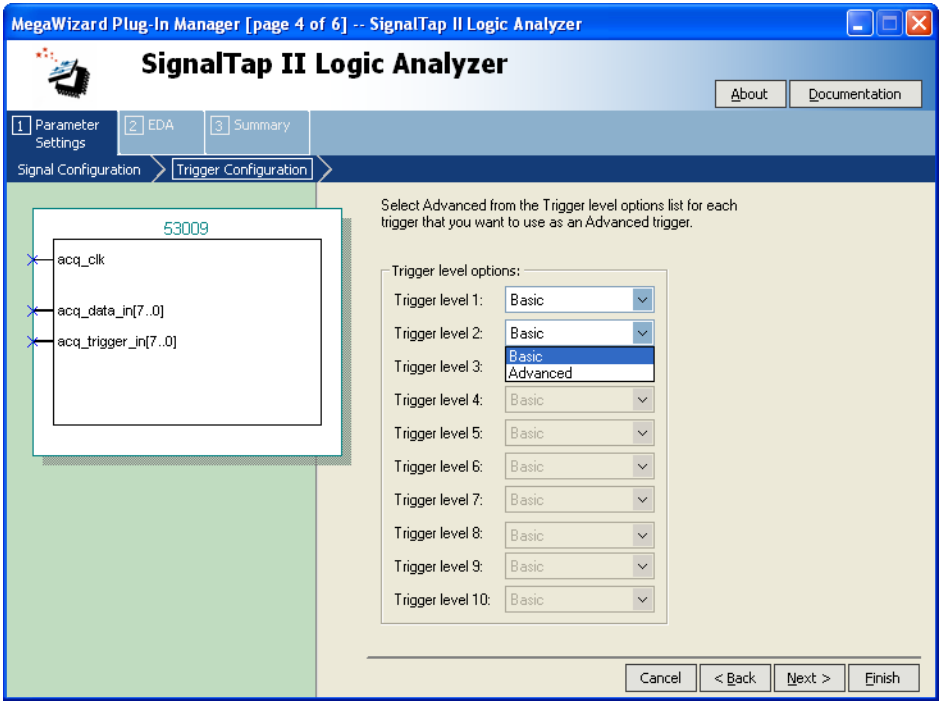
7. Click **Next**.

8. Set the **Trigger level** options by selecting **Basic** or **Advanced** (Figure 13–7). If you select **Advanced** for any trigger level, the next page of the MegaWizard Plug-In Manager displays the Advanced Trigger Condition Editor. You can configure an advanced trigger expression using the number of signals you specified for the trigger input port width.



You cannot define a Power-Up Trigger using the MegaWizard Plug-In Manager. Refer to "Define Triggers" on page 13–30 to learn how to do this using the SignalTap II file.

Figure 13–7. MegaWizard Basic and Advanced Trigger Options



9. On the final page of the MegaWizard Plug-In Manager, select any additional files you want to create and click **Finish** to create an HDL representation of the SignalTap II Logic Analyzer.

   For information about the configuration settings options in the MegaWizard Plug-In Manager, refer to "Configure the SignalTap II Logic Analyzer" on page 13–18. For information about defining triggers, refer to "Define Triggers" on page 13–30.

SignalTap II Megafunction Ports

Table 13–3 provides information about the SignalTap II megafunction ports.



For the most current information about the ports and parameters for this megafunction, refer to the latest version of the Quartus II Help.

Table 13–3. SignalTap II Megafunction Ports

Port Name	Type	Required	Description
acq_data_in	Input	No	This set of signals represents the set of signals that are monitored in the SignalTap II Logic Analyzer.
acq_trigger_in	Input	No	This set of signals represents the set of signals that are used to trigger the analyzer.
acq_clk	Input	Yes	This port represents the sampling clock that the SignalTap II Logic Analyzer uses to capture data.
trigger_in	Input	No	This signal is used to trigger the SignalTap II Logic Analyzer.
trigger_out	Output	No	This signal is enabled when the trigger event occurs.

Instantiating the SignalTap II Logic Analyzer in Your HDL

Instantiating the logic analyzer in your HDL is similar to instantiating any other Verilog HDL or VHDL megafunction in your design. Add the code from the files that are generated by the MegaWizard Plug-In Manager to your design, mapping the signals in your design to the appropriate SignalTap II megafunction ports. You can instantiate up to 127 analyzers in your design, or as many as physically fit in the FPGA. Once you have instantiated the SignalTap II file in your HDL file, compile your Quartus II project to fit the logic analyzer in the target FPGA.

To capture and view the data, you must create a SignalTap II file from your SignalTap II HDL output file. To do this, on the File menu, point to Create/Update, and click **Create SignalTap II File from Design Instance(s)**.

If you make any changes to your design or the SignalTap II instance, recreate or update the SignalTap II file with this command. This ensures that the SignalTap II file is always compatible with the SignalTap II instance in your design. If the SignalTap II file is not compatible with the SignalTap II instance in your design, you may not be able to control the SignalTap II Logic Analyzer after it is programmed into your device.

For information about SignalTap II file compatibility with programmed SignalTap II instances, refer to "Program the Target Device or Devices" on page 13–57.

Embedding Multiple Analyzers in One FPGA

The SignalTap II Logic Analyzer includes support for multiple logic analyzers in an FPGA device. This feature allows you to create a unique logic analyzer for each clock domain in the design. As multiple instances of the analyzer are added to the SignalTap II file, the resource usage increases proportionally.

In addition to debugging multiple clock domains, this feature allows you to apply the same SignalTap II settings to a group of signals in the same clock domain. For example, if you have a set of signals that must use a sample depth of 64K and another set of signals in the same clock domain requires a sample depth of 1K, you can create two instances to meet these needs.

To create multiple analyzers, on the Edit menu, click **Create Instance**, or right-click in the Instance Manager window and click **Create Instance**.

Each instance of the SignalTap II Logic Analyzer can be configured independently. The icon in the Instance Manager for the currently active instance that is available for configuration is highlighted in color and surrounded by a blue box. To configure a different instance, double-click the icon or name of another instance in the Instance Manager.

Monitoring FPGA Resources Used by the SignalTap II Logic Analyzer

The SignalTap II Logic Analyzer has a built-in resource estimator that calculates the logic resources and amount of memory that each SignalTap II Logic Analyzer uses. You can see the resource usage of each logic analyzer instance and the total resources used in the columns of the Instance Manager section of the SignalTap II Editor. This feature is useful when device resources are limited and you must know what device resources the SignalTap II Logic Analyzer uses. The value reported in the resource usage estimator may vary by as much as 5% from the actual resource usage.

Table 13–4 shows the SignalTap II Logic Analyzer M4K memory block resource usage for the listed devices per signal width and sample depth.

Table 13–4. SignalTap II Logic Analyzer M4K Block Utilization for Stratix II, Stratix, Stratix GX, and Cyclone Devices Note (1)

Signals (Width)	Samples (Depth)			
	256	512	2,048	8,192
8	< 1	1	4	16
16	1	2	8	32
32	2	4	16	64
64	4	8	32	128
256	16	32	128	512

Note to Table 13–4:

(1) When you configure a SignalTap II Logic Analyzer, the Instance Manager reports an estimate of the memory bits and logic elements required to implement the given configuration.

Configure the SignalTap II Logic Analyzer

The SignalTap II file provides many options for configuring instances of the logic analyzer. Some of the settings are similar to those found on traditional external logic analyzers. Other settings are unique to the SignalTap II Logic Analyzer because of the requirements for configuring an embedded logic analyzer. All settings give you the ability to configure the logic analyzer the way you want to help debug your design.



Some settings can only be adjusted when you are viewing Run-Time Trigger conditions instead of Power-Up Trigger conditions. To learn about Power-Up Triggers and viewing different trigger conditions, refer to "Creating a Power-Up Trigger" on page 13–45.

Assigning an Acquisition Clock

You must assign a clock signal to control the acquisition of data by the SignalTap II Logic Analyzer. The logic analyzer samples data on every rising edge of the acquisition clock. You can use any signal in your design as the acquisition clock. However, for best results, Altera recommends that you use a global, non-gated clock for data acquisition. Using a gated clock as your acquisition clock can result in unexpected data that does not accurately reflect the behavior of your design. The Quartus II Classic Timing Analyzer shows the maximum acquisition clock frequency at which you can run your design.

To assign an acquisition clock, perform the following steps:

1. In the SignalTap II Logic Analyzer window, click the **Setup** tab.
2. Click **Browse** next to the **Clock** field in the Signal Configuration pane. The **Node Finder** dialog box appears.
3. From the **Filter** list, select **SignalTap II: post-fitting**
   or
   SignalTap II: pre-synthesis.
4. In the **Named** field, type the exact name of a node that you want to use as your sample clock, or search for a node using a partial name and wildcard characters.
5. To start the node search, click **List**.
6. In the **Nodes Found** list, select the node that represents the design's global clock signal.
7. Add the selected node name to the **Selected Nodes** list by clicking ">" or by double-clicking the node name.
8. Click **OK**. The node is now specified as the acquisition clock in the SignalTap II Editor.

If you do not assign an acquisition clock in the SignalTap II Editor, the Quartus II software automatically creates a clock pin called `auto_stp_external_clk`.

You must make a pin assignment to this pin independently from the design. You must ensure that a clock signal in your design drives the acquisition clock.



For information about assigning signals to pins, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Adding Signals to the SignalTap II File

While configuring the logic analyzer, you add signals to the node list in the SignalTap II file to select which signals in your design you want to monitor. Selected signals are also used to define triggers. You can assign the following two types of signals to your SignalTap II file:

- **Pre-synthesis**—This signal exists after design elaboration, but before any synthesis optimizations are done. This set of signals should reflect your Register Transfer Level (RTL) signals.

- **Post-fitting**—This signal exists after physical synthesis optimizations and place-and-route.



If you are not using incremental compilation, add only pre-synthesis signals to your SignalTap II file. Using pre-synthesis is particularly useful if you want to add a new node after you have made design changes. To do this, on the Processing menu, point to Start and click **Start Analysis & Elaboration**.

Signals shown in blue text are post-fit node names. Signals shown in black text are pre-synthesis node names.

After successful Analysis and Elaboration, the signals shown in red text are invalid signals. Unless you are certain that these signals are valid, you must remove them from the SignalTap II file for correct operation. The SignalTap II Health Monitor also indicates if an invalid node name exists in the SignalTap II file.

As a general guideline, signals can be tapped if a routing resource (row or column interconnects) exists to route the connection to the SignalTap II instance. For example, signals that exist in the I/O element (IOE) cannot be directly tapped because there are no direct routing resources from the signal in an IOE to a core logic element. For input pins, you can tap the signal that is driving a Logic Array Block (LAB) from an IOE, or, for output pins, you can tap the signal from the LAB that is driving an IOE.

When adding pre-synthesis signals, all connections made to the SignalTap II Logic Analyzer are made prior to synthesis. Logic and routing resources are allocated during recompilation to make the connection as if a change in your design files had been made. As such, pre-synthesis signal names for signals driving to and from IOEs will coincide with the signal names assigned to the pin.

In the case of post-fit signals, connections that you make to the SignalTap II Logic Analyzer are the signal names from the actual atoms in your post-fit netlist. A connection can only be made if the signals are part of the existing post-fit netlist and existing routing resources are available from the signal of interest to the SignalTap II Logic Analyzer. In the case of post-fit output signals, tap the `COMBOUT` or `REGOUT` signal that drives the IOE block. For post-fit input signals, signals driving into the core logic will coincide with the signal name assigned to the pin.



If you are tapping the signal from the atom that is driving an IOE, be aware that the signal may be inverted due to NOT-gate push back. You can check this by locating the signal in either the Resource Property Editor or the Technology Map Viewer. The Technology Map viewer and the Resource Property Editor are also helpful in finding post-fit node names.



For information about cross-probing to source design file and other Quartus II windows, refer to the *Analyzing Designs with Quartus II Netlist Viewers* chapter in volume 1 of the *Quartus II Handbook*.

For more information about the use of incremental compilation with the SignalTap II Logic Analyzer, refer to "Faster Compilations with Quartus II Incremental Compilation" on page 13–51.

Signal Preservation

Many of your RTL signals are optimized during the process of synthesis and place-and-route. The RTL signal names frequently may not appear in the post-fit netlist after optimizations. This can cause a problem when you use the incremental compilation flow with the SignalTap II Logic Analyzer. Since only post-fitting signals can be added to the SignalTap II Logic Analyzer in partitions of type **post-fit**, RTL signals that you want to monitor may not be available, preventing their usage. To avoid this issue, you can use synthesis attributes to preserve signals during synthesis and place-and-route. When the Quartus II software encounters these synthesis attributes, it does not perform any optimization on the specified signals, forcing them to continue to exist in the post-fit netlist. However, if you do this, you could see an increase in resource utilization or a decrease in timing performance. The two attributes you can use are:

- `keep`—Ensures that combinational signals are not removed
- `preserve`—Ensures that registers are not removed



For more information about using these attributes, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook.*

If you are debugging an IP core, such as the Nios II CPU, or other encrypted IP, you may need to preserve nodes from the core to make them available for debugging with the SignalTap II Logic Analyzer. This is often necessary when a plug-in is used to add a group of signals for a particular IP. To do this, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**. Turn on **Create debugging nodes for IP cores** to make these nodes available to the SignalTap II Logic Analyzer.

Assigning Data Signals

To assign data signals, perform the following steps:

1. Perform Analysis and Elaboration, Analysis and Synthesis, or compile your design.
2. In the SignalTap II Logic Analyzer window, click the **Setup** tab.
3. Double-click anywhere in the node list of the SignalTap II Editor to open the **Node Finder** dialog box.
4. In the **Fitter** list, select **SignalTap II: pre-synthesis** or **SignalTap II: post-fitting**. Only signals listed under one of these filters can be added to the SignalTap II node list. Signals cannot be selected from any other filters.

   If you use Incremental Compilation flow with SignalTap II, pre-synthesis nodes will not be connected to the SignalTap II Logic Analyzer if the affected partition is of the post-fit type. Any pre-synthesis nodes added to a partition of type post-fit may not be connected to the SignalTap II Logic Analyzer. A critical warning is issued for all pre-synthesis node names that are not found in the post-fit netlist. Altera recommends that you do not add a mix of pre-synthesis and post-fitting signals within the same partition. For more details, refer to "Using Incremental Compilation with the SignalTap II Logic Analyzer" on page 13–52.
5. In the **Named** field, type a node name, or search for a particular node by entering a partial node name along with wildcard characters. To start the node name search, click **List.**
6. In the **Nodes Found** list, select the node or bus you want to add to the SignalTap II file.
7. Add the selected node name(s) to the **Selected Nodes** list by clicking ">" or by double-clicking the node name(s).
8. To insert the selected nodes in the SignalTap II file, click **OK**. With the default colors set for the SignalTap II Logic Analyzer, a pre-synthesis signal in the list is shown in black, and a post-fitting signal is shown in blue.

   You can also drag and drop signals from the **Node Finder** dialog box into a SignalTap II file.

Node List Signal Use Options

Once a signal is added to the node list, you can select options that specify how the signal is used with the logic analyzer. You can turn off the ability of a signal to trigger the analyzer by disabling the **Trigger Enable** for that signal in the node list in the SignalTap II file. This option is useful when you want to see only the captured data for a signal, and you are not using that signal as part of a trigger.

You can turn off the ability to view data for a signal by disabling the **Data Enable** column. This option is useful when you want to trigger on a signal, but have no interest in viewing data for that signal.

For information about using signals in the node list to create SignalTap II trigger conditions, refer to "Define Triggers" on page 13–30.

Untappable Signals

Not all of the post-fitting signals in your design are available in the **SignalTap II: post-fitting** filter in the **Node Finder** dialog box. The following signal types cannot be tapped:

- Post-fit output pins—You cannot tap a post-fit output pin directly. To make an output signal visible, tap the register or buffer that drives the output pin.
- Signals that are part of a carry chain—You cannot tap the carry out (`cout0` or `cout1`) signal of a logic element. Due to architectural restrictions, the carry out signal can only feed the carry in of another logic element (LE).
- JTAG Signals—You cannot tap the JTAG control (`TCK`, `TDI`, `TDO`, and `TMS`) signals.
- `altgxb` megafunction—You cannot directly tap any ports of an `altgxb` instantiation.
- LVDS—You cannot tap the data output from a serializer/deserializer (SERDES) block.

Adding Signals with a Plug-In

Instead of adding individual or grouped signals through the **Node Finder**, you can add groups of relevant signals of a particular type of IP through the use of a plug-in. The SignalTap II Logic Analyzer comes with one plug-in already installed for the Nios II processor. Besides easy signal addition, plug-ins also provide a number of other features, such as pre-designed mnemonic tables, useful for trigger creation and data viewing, as well as the ability to disassemble code in captured data.

The Nios II plug-in, for example, creates one mnemonic table in the **Setup** tab, and two tables in the **Data** tab:

- **Nios II Instruction** (**Setup** tab) —Capture all the required signals for triggering on a selected instruction address.
- **Nios II Instance Address** (**Data** tab)—Display address of executed instructions in hexadecimal format or as a programming symbol name if defined in an optional Executable and Linking Format (**.elf**) file.
- **Nios II Disassembly** (**Data** tab)—Displays disassembled code from the corresponding address.

For information about the other features plug-ins provided, refer to "Define Triggers" on page 13–30 and "View, Analyze, and Use Captured Data" on page 13–63.

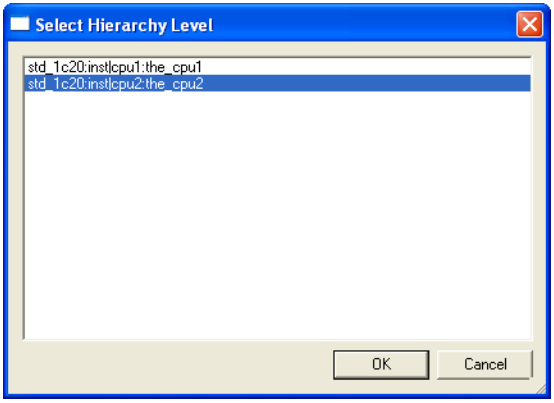
To add signals to the SignalTap II file using a plug-in, perform the following steps after running Analysis and Elaboration on your design:

1. Right-click in the node list. On the **Add Nodes with Plug-In** submenu, click the name of the plug-in you want to use, such as the included plug-in named **Nios II**.

   If the intellectual property (IP) for the selected plug-in does not exist in your design, a message appears informing you that you cannot use the selected plug-in.

2. The **Select Hierarchy Level** dialog box appears showing the IP hierarchy of your design (Figure 13–8). Select the **IP** that contains the signals you want to monitor with the plug-in, and click **OK**.

Figure 13–8. IP Hierarchy Selection



3. If all the signals in the plug-in are available, a dialog box may appear, depending on the plug-in selected, where you can set any available options for the plug-in. With the Nios II plug-in, you can optionally select an Executable and Linking Format (**.elf**) file containing program symbols from your Nios II Integrated Development Environment (IDE) software design. Set options for the selected plug-in as desired, and click **OK**.



To make sure all the required signals are available, turn on the **Create debugging nodes for IP cores** option in the Quartus II Analysis & Synthesis settings.

All the signals included in the plug-in are added to the node list.

Specifying the Sample Depth

The sample depth specifies the number of samples that are captured and stored for each signal in the captured data buffer. To set the sample depth, select the desired number of samples to store in the **Sample Depth** list. The sample depth ranges from 0 to 128K.

If device memory resources are limited, you may not be able to successfully compile your design with the sample buffer size you have selected. Try reducing the sample depth to reduce resource usage.

Capturing Data to a Specific RAM Type

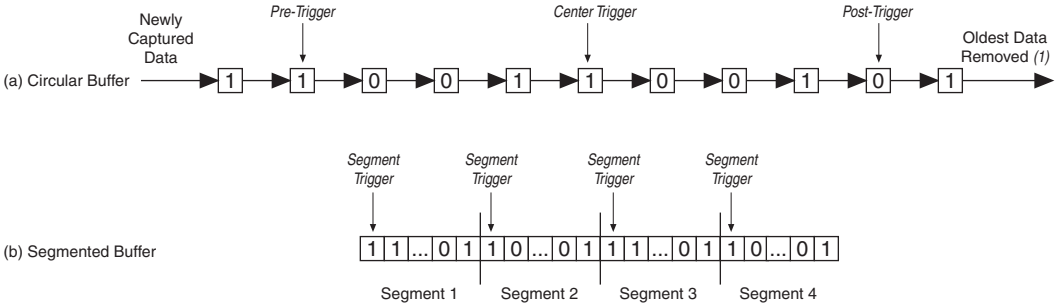
When you use the SignalTap II Logic Analyzer with some devices, you have the option to select the RAM type where acquisition data is stored. RAM selection allows you to preserve a specific memory block for your design and allocate another portion of memory for SignalTap II data acquisition. For example, if your design implements a large buffering application such as a system cache, it is ideal to place this application into M-RAM blocks so that the remaining M512 or M4K blocks are used for SignalTap II data acquisition.

To select the RAM type to use for the SignalTap II buffer, select it from the **RAM type** list. Use this feature when the acquired data (as reported by the SignalTap II resource estimator) is not larger than the available memory of the memory type that you have selected in the FPGA.

Choosing the Buffer Acquisition Mode

The buffer acquisition type selection feature in the SignalTap II Logic Analyzer lets you choose how the captured data buffer is organized and can potentially reduce the amount of memory that is required for SignalTap II data acquisition. You can choose to use either a circular buffer, which allocates the entire sample depth to a single buffer, or a segmented buffer, which splits the buffer space into a number of separate even sized segments. Figure 13–9 illustrates the differences between the two buffer types.

Figure 13–9. Buffer Type Comparison in the SignalTap II Logic Analyzer *Note (1)*



Note to Figure 13–9:

(1) Both circular and segmented buffers can use a predefined trigger position or define a custom trigger position using the **State-Based Triggering** tab. Refer to "Specifying the Trigger Position" on page 13–44 for more details.

Circular Buffer

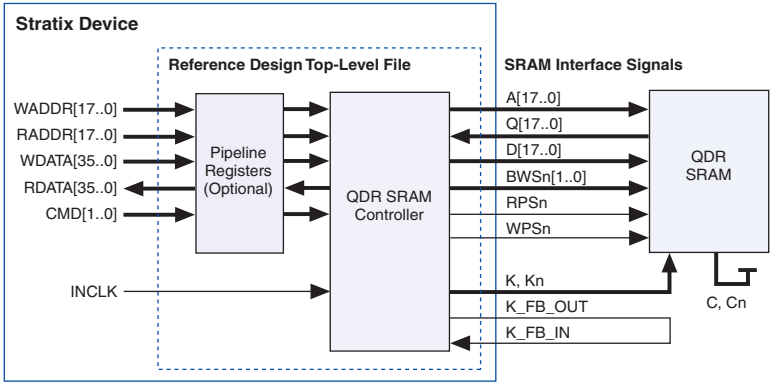
The circular buffer (Figure 13–9 (a)) is the default buffer type used by the SignalTap II Logic Analyzer. While the logic analyzer is running, data is stored in the buffer until it fills up, at which point new data replaces the oldest data. This continues until a specified trigger event occurs. When this happens, the logic analyzer continues to capture data after the trigger event until the buffer is full, based on the circular buffer trigger position setting in the Signal Configuration pane in the SignalTap II file. Select a setting from the list to choose whether to capture the majority of the data before (**Post trigger position**) or after (**Pre trigger position**) the trigger occurs or to center the trigger position in the data (**Center trigger position**). Another option is to use the custom state-based triggering flow to define your desired triggering position precisely. You can also choose to continuously capture data until the logic analyzer is stopped.

For more information, refer to "Specifying the Trigger Position" on page 13–44.

Segmented Buffer

The segmented buffer (Figure 13–9 (b)) organizes the buffer into a number of separate, evenly sized segments. This type of buffer organization makes it easier to debug systems that contain relatively infrequent recurring events. Figure 13–10 shows an example of this type of buffer system.

Figure 13–10. Example System that Generates Recurring Events



The SignalTap II Logic Analyzer verifies the functionality of the design shown in Figure 13–10 to ensure that the correct data is written to the SRAM controller. The buffer acquisition in the SignalTap II Logic

Analyzer allows you to monitor the RDATA port when H'0F0F0F0F is sent into the RADDR port. You can monitor multiple read transactions from the SRAM device without running the SignalTap II Logic Analyzer again. The buffer acquisition feature allows you to segment the memory so that you can capture the same event multiple times without wasting the allocated memory. The number of cycles that are captured depends on the number of segments that you have specified under the **Data** settings.

To enable and configure buffer acquisition, select **Segmented** in the SignalTap II Editor, and select the number of segments to use. In the example, selecting sixty-four, 64-sample segments allows you to capture 64 read cycles when the RADDR signal is H'0F0F0F0F.

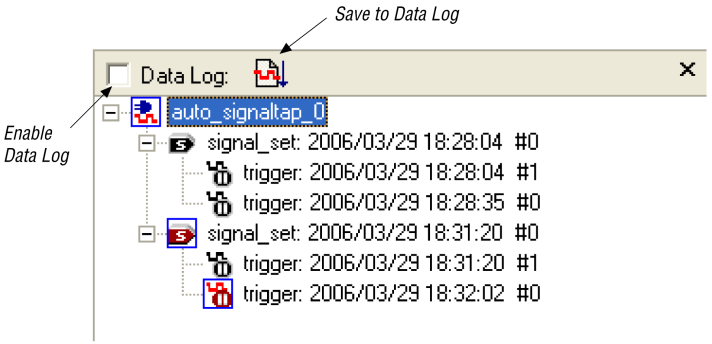
For more information about the buffer acquisition mode, refer to *Setting the Buffer Acquisition Mode* in the Quartus II Help.

Managing Multiple SignalTap II Files and Configurations

In some cases you may have more than one SignalTap II file in one design. Each file potentially has a different group of monitored signals. These signal groups make it possible to debug different blocks in your design. In turn, each group of signals may also be used to define different sets of trigger conditions. Along with each SignalTap II file, there is also an associated programming file (SRAM Object File (SOF)). The settings in a selected SignalTap II file must match the SignalTap II logic design in the associated SOF file for the logic analyzer to run properly when the device is programmed. Managing all of the SignalTap II files and their associated settings and programming files is a challenging task. To help you manage everything, you can use the **Data Log** feature and the **SOF Manager**.

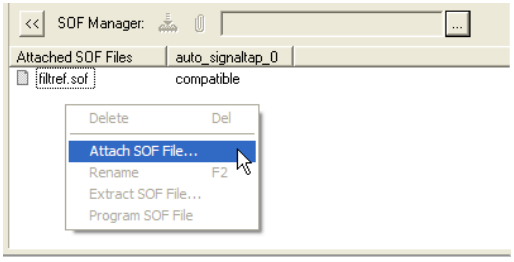
The **Data Log** allows you to store multiple SignalTap II configurations within a single SignalTap II file. Figure 13–11 shows two signal set configurations with multiple trigger conditions in one SignalTap II file. To toggle between the active configurations, double-click on an entry in the **Data Log**. As you toggle between the different configurations, the signal list and trigger conditions change in the **Setup** tab of the SignalTap II file. The active configuration displayed in the SignalTap II file is indicated by the blue square around the signal set in the **Data log**. To store a configuration in the data log, on the Edit menu, click **Save to Data Log,** or click the **Save to Data Log** button at the top of the Data Log.

Figure 13–11. Data Log



The SOF Manager allows you to embed multiple SOFs into one SignalTap II file. Embedding an SOF in a SignalTap II file lets you move the SignalTap II file to a different location, either on the same computer or across a network, without the need to include the associated SOF as a separate file. To embed a new SOF in the SignalTap II file, right-click in the SOF Manager, and click **Attach SOF File** (Figure 13–12).

Figure 13–12. SOF Manager



As you switch between configurations in the Data Log, you can extract the SOF that is compatible with that particular configuration and use the programmer in the SignalTap II Logic Analyzer to download the new SOF to the FPGA. In this way, you ensure that the configuration of your SignalTap II file always matches the design programmed into the target device.

Define Triggers

To capture the data you want at the right time, you need to specify conditions under which the signals you are monitoring display that data. In the SignalTap II Logic Analyzer, these conditions are referred to as triggers, just as they are in conventional external logic analyzers and oscilloscopes. You have many options for creating different types of triggers to help in your debugging.

Creating Basic Trigger Conditions

The simplest kind of trigger condition you can use is a basic trigger. You select this from the list at the top of the **Trigger Conditions** column in the node list in the SignalTap II Editor. With the trigger type set to **Basic**, you must set the trigger pattern for each signal you have added in the SignalTap II file. To set the trigger pattern, right-click in the **Trigger Conditions** column and click the desired pattern. You can set the trigger pattern to any of the following conditions:

- Don't Care
- Low
- High
- Falling Edge
- Rising Edge
- Either Edge

For buses, you can type a pattern in binary, or right-click and select **Insert Value** to enter the pattern in other number formats. For signals added to the SignalTap II file that have an associated mnemonic table, you can right-click and select an entry from the table to set pre-defined conditions for the trigger.

For more information about the creation and use of mnemonic tables, refer to "View, Analyze, and Use Captured Data" on page 13–63 and in the Quartus II Help.

For signals added with certain plug-ins, you can easily create basic triggers using pre-defined mnemonic table entries. For example, with the Nios II plug-in, if you have specified an executable software (**.elf**) file from your Nios II IDE design, you can type the name of a function from your Nios II code. The logic analyzer triggers when the Nios II instruction address matches the address of the specified code function name.

Data capture stops and the data is stored in the buffer when the logical `AND` of all the signals for a given trigger condition evaluates to `TRUE`.

Creating Advanced Trigger Conditions

Along with the SignalTap II Logic Analyzer's basic triggering capabilities, you can build more complex triggers utilizing extra logic that enable you to capture data when a particular combination of conditions exist. If you set the trigger type to **Advanced** at the top of the **Trigger Conditions** column in the node list of the SignalTap II Editor, a new tab named **Advanced Trigger** appears where you can build a complex trigger expression using a simple GUI. You can drag and drop operators into the Advanced Trigger Configuration Editor window to build the complex trigger condition in an expression tree. Double-click operators that you have placed or right-click them and select Properties to configure the operator's settings. Table 13–5 lists the operators you can use.

Table 13–5. Advanced Triggering Operators *Note (1)*

Name of Operator	Type
Less Than	Comparison
Less Than or Equal To	Comparison
Equality	Comparison
Inequality	Comparison
Greater Than	Comparison
Greater Than or Equal To	Comparison
Logical NOT	Logical
Logical AND	Logical
Logical OR	Logical
Logical XOR	Logical
Reduction AND	Reduction
Reduction OR	Reduction
Reduction XOR	Reduction
Left Shift	Shift
Right Shift	Shift
Bitwise Complement	Bitwise
Bitwise AND	Bitwise
Bitwise OR	Bitwise
Bitwise XOR	Bitwise
Edge and Level Detector	Signal Detection

Note to Table 13–5:
(1) For more information about each of these operators, refer to the Quartus II Help.

You can configure some of the settings for certain operators at run-time. This enables you to change one operator type to another operator type or adjust other settings for an operator without recompiling your design. Operator settings that have a white background on the operator symbol can be changed without recompiling the design.

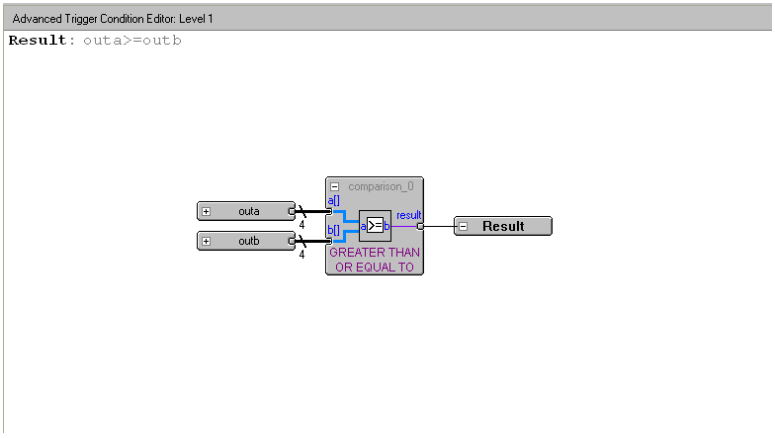
Adding many objects to the Advanced Trigger Condition Editor can make the workspace cluttered and difficult to read. To keep objects organized while you build your advanced trigger condition, use the right-click menu and select **Arrange All Objects**. You can also use the **Zoom-Out** command to fit more objects into the Advanced Trigger Condition editor window.

Examples of Advanced Triggering Expressions

The following examples show how to use Advanced Triggering:

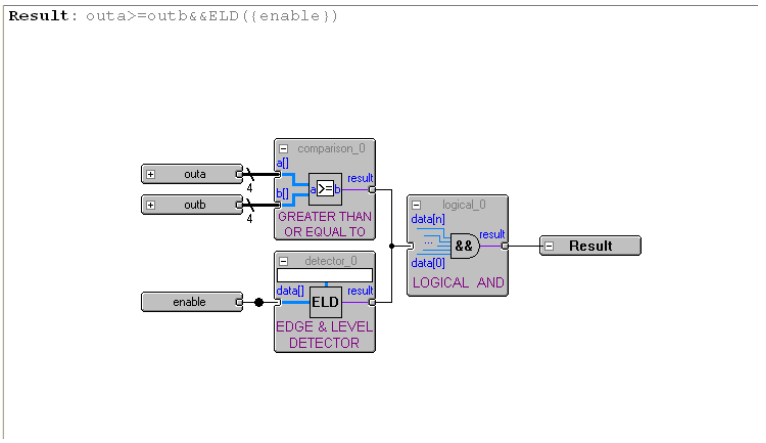
- Trigger when bus `outa` is greater than or equal to `outb` (Figure 13–13).

Figure 13–13. Bus outa is Greater Than or Equal to Bus outb



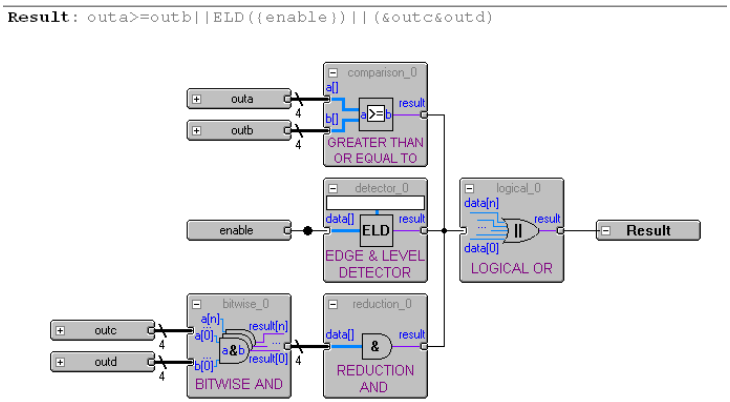
- Trigger when bus `outa` is greater than or equal to bus `outb`, and when the enable signal has a rising edge (Figure 13–14).

Figure 13–14. Enable Signal has a Rising Edge



- Trigger when bus `outa` is greater than or equal to bus `outb`, or when the enable signal has a rising edge. Or, when a bitwise `AND` operation has been performed between bus `outc` and bus `outd`, and all bits of the result of that operation are equal to 1 (Figure 13–15).

Figure 13–15. Bitwise AND Operation



Trigger Condition Flow Control

SignalTap II offers multiple triggering conditions to give you more precise control of the method in which data is captured into the acquisition buffers. Trigger Condition Flow control allows you to define the relationship between a set of triggering conditions. SignalTap II gives you two flow control mechanisms for organizing trigger conditions:

- Sequential Triggering—The default triggering flow. This flow allows you to define up to ten triggering levels that must be satisfied before the acquisition buffer finishes capturing.
- Custom State-Based Triggering—This flow allows you the greatest control over your acquisition buffer. This method allows you to organize trigger conditions into states based on a conditional flow that you define.

Both methods can be used with either a circular buffer or a segmented buffer.

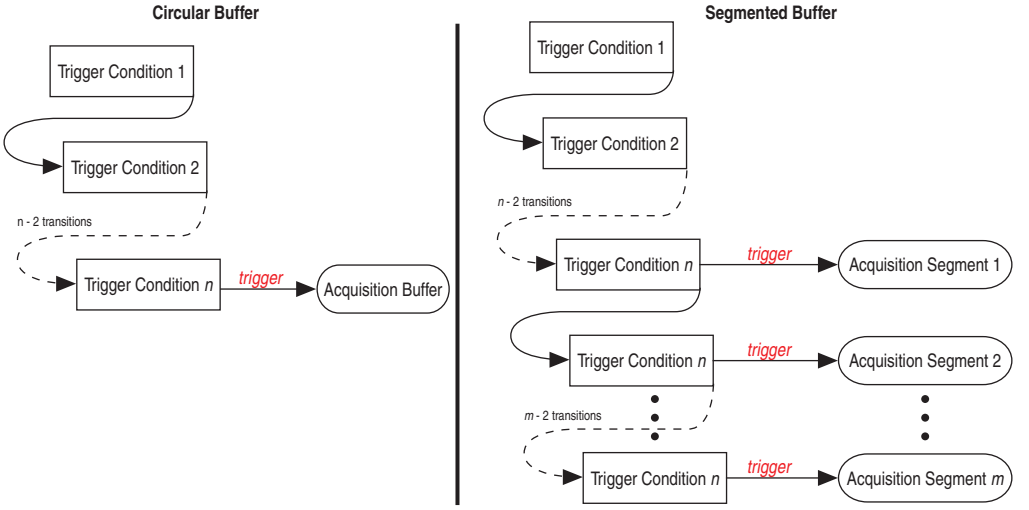
Sequential Triggering

The sequential triggering flow allows you to cascade up to ten levels of triggering conditions. The SignalTap II Logic Analyzer sequentially evaluates each of the triggering conditions. When the last triggering condition evaluates to TRUE, the SignalTap II Logic Analyzer triggers the acquisition buffer. For segmented buffers, every acquisition segment after the first segment triggers on the last triggering condition that you have specified. You can use the simple sequential triggering feature with basic triggers, advanced triggers, or a mix of both. Figure 13–16 illustrates the simple sequential triggering flow for circular and segmented buffers.



Note that the external trigger in is considered as trigger level `0`. The external trigger must be evaluated before the main trigger levels are evaluated.

Figure 13–16. Sequential Triggering Flow *Notes (1), (2)*



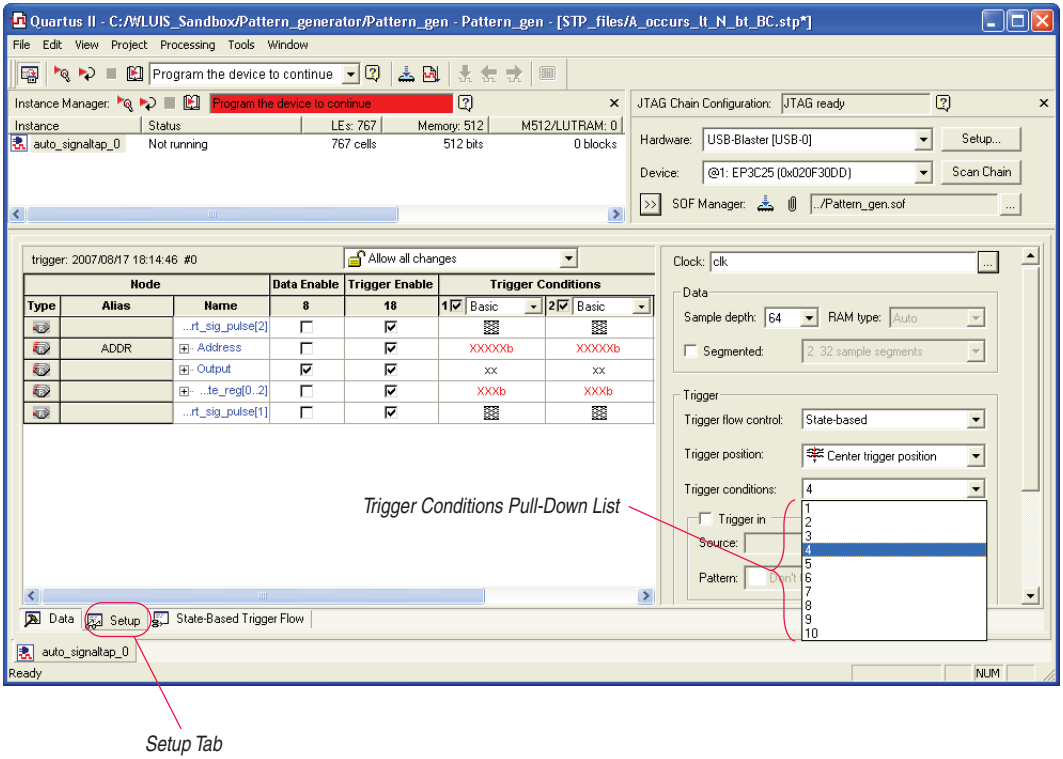
Note to Figure 13–16:

(1) The Acquisition buffer stops capture when all n triggering levels are satisfied, where $n \leq 10$.

(2) An external trigger input, if defined, will be evaluated before all other defined trigger conditions are evaluated. For more information about external triggers refer to "Using External Triggers" on page 13–47.

To configure the SignalTap II Logic Analyzer for Sequential triggering, on the Trigger flow control list in the SignalTap II editor, select **Sequential**. You can select the desired number of trigger conditions by using the **Trigger Conditions** pull-down list. After you select the desired number of trigger conditions, you can configure each trigger condition in the node list. To disable any trigger condition, click the check box next to the trigger condition at the top of the column in the node list. Figure 13–17 shows the setup tab for Sequential Triggering.

Figure 13–17. Setup Tab

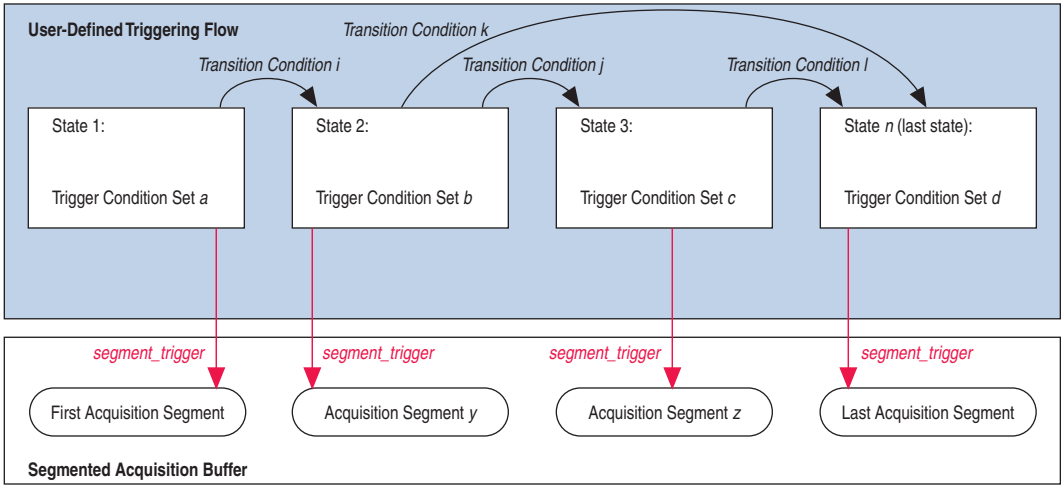


Custom State-Based Triggering

The custom state-based triggering method gives you the most control of triggering condition arrangement. This flow gives you the ability to describe the relationship between triggering conditions precisely, using an intuitive GUI and the SignalTap II Trigger Flow Description Language, a simple description language based upon conditional expressions. Tooltips within the custom triggering flow GUI allow you to describe your desired flow quickly. The custom state-based triggering flow allows for more efficient use of the space available in the acquisition buffer because only specific samples of interest are captured.

Figure 13–18 illustrates the custom state-based triggering flow. Events that trigger the acquisition buffer are organized by a user-defined state diagram. All actions performed by the acquisition buffer are captured by the states and all the transition conditions between the states are defined by the conditional expressions that you specify within each state.

Figure 13–18. Custom State-Based Triggering Flow *Note (1), (2)*



Note to Figure 13–18:
(1) You are allowed up to twenty different states.
(2) An external trigger input, if defined, will be evaluated before any conditions in the custom state-based triggering flow are evaluated. For more information, refer to "Using External Triggers" on page 13–47.

Each state allows you to define a set of conditional expressions. Each conditional expression is a Boolean expression dependent upon a combination of triggering conditions (configured within the **Setup** tab), counters, and status flags. Counters and status flags are resources provided by the Signal Tap II custom-based triggering flow.

Within each conditional expression you define a set of "actions". Actions include triggering the acquisition buffer to stop capture, a modification to either a counter or status flag, or a state transition.

Trigger actions can apply to either a single segment of a segmented acquisition buffer or to the entire circular acquisition buffer. Each trigger action provides you with an optional count that specifies the number of samples to be captured before stopping acquisition of the current segment. The count argument allows you to control the amount of data captured precisely before and after triggering event.

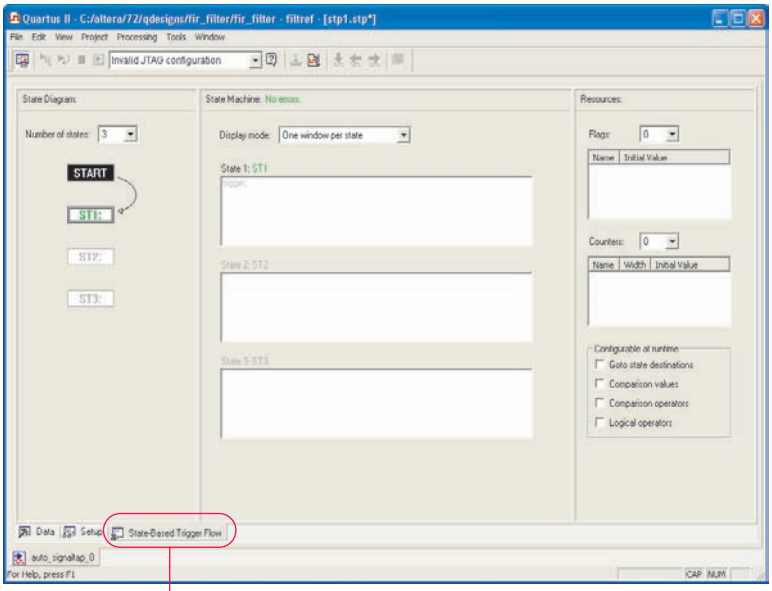
Resource manipulation actions allow you to increment and decrement counters or set and clear status flags. The counter and status flag resources are used as optional inputs in the conditional expressions. Counters and status flags are useful for counting the number of occurrences of particular events and for aiding in the triggering flow control.

This Signal Tap II custom state-based triggering flow allows you to capture a sequence of events that may not necessarily be contiguous in time; for example, capturing a communication transaction between two devices that includes a handshaking protocol containing a sequence of acknowledgements.

The **State-Based Trigger Flow** tab is the control interface for the custom state-based triggering flow. To enable this tab, on the Trigger Flow Control pull-down list, select **State-based**. (Note that when the **Trigger Flow control** option is set to **Sequential**, the **State-Based Trigger Flow** tab is hidden.)

Figure 13–19 shows the **Custom Trigger Flow** tab.

Figure 13–19. State-Based Trigger Flow Tab



The State-Based Trigger Flow tab is partitioned into the following three panes.

- State Diagram Pane
- Resources Pane
- State Machine Pane

State Diagram Pane
The State Diagram pane provides a graphical overview of the triggering flow that you define. It shows the number of states available and the state transitions between all of the states. You can adjust the number of available states by using the pull-down menu above the graphical overview.

State Machine Pane
The State Machine pane contains the text entry boxes where you can define the triggering flow and the actions associated with each state. You can define the triggering flow using the Signal Tap II Trigger Flow Description Language, a simple language based upon if-else conditional statements. Tooltips appear when you move the mouse over the cursor, to guide command entry into the state boxes. The GUI provides a syntax check on your flow description in real-time and highlights any errors in the text flow.



Refer to "SignalTap II Trigger Flow Description Language" on page 13–40 for a full description of the SignalTap II Trigger Flow Description Language. You can also refer to the Quartus II Help.

The State Machine description text boxes default to show one text box per state. You can optionally have the entire flow description be shown in a single text field. This option can be useful when copying and pasting a flow description from a template or an external text editor. To toggle between one window per state, or all states in one window, select the appropriate option under **State Display mode.**

Resources Pane
The Resources pane allows you to declare Status Flags and Counters for use in the conditional expressions in the Custom Triggering Flow. Actions to decrement and increment counters or to set and clear status flags are performed within the triggering flow that you define.

You can set up to 20 counters and 20 status flags for use. Counter and status flags values may be initialized by right-clicking the status flag or counter name after selecting a number of them from the respective drop-down list, and selecting **Set Initial Value.** Counter width can be set by right-clicking the counter name and selecting **Set Width**.

Runtime Reconfigurability—The **configurable at runtime** options in the Resources pane allows you to configure the custom-flow control options that can be changed at runtime without requiring a recompilation. Table 13–6 contains a description of options that can be reconfigured at runtime.

Table 13–6. Runtime Configurable Settings

Setting	Description
Destination of goto action	Allows you to modify the destination of the state transition at runtime.
Comparison values	Allows comparison values in Boolean expressions to be modifiable at runtime. In addition, it allows the `segment_trigger` and trigger action post-fill count argument to be modifiable at runtime.
Comparison operators	Allows comparison operators in Boolean expressions to be modifiable at runtime.
Logical operators	Allows the logical operators in Boolean expressions to be modifiable at runtime.

You can restrict changes to your SignalTap configuration to include only the options that do not require a recompilation by using the pull-down menu above the trigger list in the **Setup** tab. The option **Allow trigger condition changes only** restricts changes to only the configuration settings that the runtime configurable option set. You can then modify Trigger Flow conditions in the **Custom Trigger Flow** tab by clicking the desired parameter in the text box, and selecting a new parameter from the menu that appears.



The runtime configurable settings for the **Custom Trigger Flow** tab are on by default. You may get some performance advantages by disabling some of the runtime configurable options. Refer to "Performance and Resource Considerations" on page 13–55 for details about the effects of turning off the runtime modifiable options.

SignalTap II Trigger Flow Description Language

The Trigger Flow Description Language is based on a list of conditional expressions per state to define a set of actions. Each line in Example 13–1 shows a language format. Keywords are shown in bold**.** Non-terminals are delimited by "<>" and are further explained in the following sections. Optional arguments are delimited by " [] ".



Examples of Triggering Flow descriptions for common scenarios using the Signal Tap II Custom Triggering Flow are provided in the section, "Custom Triggering Flow Application Examples" on page 13–77.

Example 13–1. Trigger Flow Description Language Format *Note (1)*

```
state <State_label>:
<action_list>

or

state <State_label>:
if( <Boolean_expression> )
<action_list>
[else if ( <boolean_expression> )
   <action_list>] (1)
[else
   <action_list>]
```

Notes to Example 13–1:
(1) Multiple `else if` conditions are allowed.

The priority for evaluation of conditional statements is assigned from top to bottom. The <*boolean_expression*> in an `if` statement can contain a single event, or it can contain multiple event conditions. The `action_list` embedded within an `if` or an `else if` clause must be delimited by the `begin` and `end` tokens when the action list contains multiple statements. When the boolean expression is evaluated true, the logic analyzer analyzes all of the commands in the action list concurrently. The possible actions include:

- Triggering the acquisition buffer
- Manipulating a counter or status flag resource
- Defining a state transition

State Labels

State Labels are identifiers that can be used in the action `goto`.

`state` <*state_label*>`:` begins the description of the actions evaluated when this state is reached.

The description of a state ends with the beginning of another state or the end of the whole trigger flow description.

Boolean_expression

`Boolean_expression` is a collection of logical operators, relational operators, and their operands that evaluate into a Boolean result. Depending on the operator, the operand can be a reference to a trigger condition, a counter and a register, or a numeric value. Within an expression, parentheses can be used to group a set of operands.

Logical operators accept any boolean expression as an operand. The supported Logical operators are shown in Table 13–7.

Table 13–7. Logical Operators

Operator	Description	Syntax
!	NOT operator	! expr1
&&	AND operator	expr1 && expr2
||	OR operator	expr1 || expr2

Relational operators are performed on counters or status flags. The comparison value—the right operator—must be a numerical value. The supported Relational operators are shown in Table 13–8.

Table 13–8. Relational Operators

Operator	Description	Syntax *Notes (1) (2)*
>	Greater than	<*identifier*> > <*numerical_value*>
>=	Greater than or Equal to	<*identifier*> >= <*numerical_value*>
==	Equals	<*identifier*> == <*numerical_value*>
!=	Does not equal	<*identifier*> != <*numerical_value*>
<=	Less than or equal to	<*identifier*> <= <*numerical_value*>
<	Less than	<*identifier*> < <*numerical_value*>

Notes to Table 13–8:
(1) <*identifier*> indicates a counter or status flag
(2) <*numerical_value*> indicates an integer

Action_list

Action_list is a list of actions that can be performed when a state is reached and a condition is also satisfied. If more than one action is specified, they must be enclosed by `begin` and `end`. The actions can be categorized as resource manipulation actions, buffer control actions and state transition actions. Actions must be embedded within a condition statement if condition statements are used in a state. Each action is terminated by a semicolon.

Resource Manipulation Action

The resources used in the trigger flow description can be either counters or status flags. Table 13–9 shows the description and syntax of each action.

Table 13–9. Resource Manipulation Action

Action	Description	Syntax
Increment	Increments a counter resource by 1	`increment <counter_identifier>;`
Decrement	Decrements a counter resource by 1	`decrement <counter_identifier>;`
Reset	Resets counter resource to initial value	`reset <counter_identifier>;`
Set	Sets a status Flag to 1	`set <register_flag_identifier>;`
Clear	Sets a status Flag to 0	`clear <register_flag_identifier>;`

Buffer Control Action

Buffer control actions specify an action to control the acquisition buffer. Table 13–10 shows the description and syntax of each action,

Table 13–10. Buffer Control Action

Action	Description	Syntax
trigger	Stops the acquisition for the current buffer and ends analysis. This command is required in every flow definition.	`trigger <post-fill_count>;`
segment_trigger	Ends the acquisition of the current segment. The Signal Tap II Logic Analyzer starts acquiring from the next segment upon evaluating this command. If all segments are filled, the oldest segment is overwritten with the latest sample. The acquisition stops when a trigger action is evaluated. This action cannot be used in non-segmented acquisition mode.	`segment_trigger <post-fill_count>;`

Both `trigger` and `segment_trigger` actions accept an optional post-fill count argument. If provided, the current acquisition acquires the number of samples provided by post-fill count and then stops acquisition. If no post-count value is specified, the trigger position for the affected buffer defaults to the trigger position specified in the setup tab.



Note that in the case of `segment_trigger`, acquisition of the current buffer stops immediately if a subsequent triggering action is issued in the next state, regardless of whether or not the post-fill count has been satisfied for the current buffer. The remaining unfilled post-count acquisitions in the current buffer are discarded and displayed as grayed-out samples in the data window.

State Transition Action

State transition action specifies the next state in the custom state control flow. It is specified by the `goto` command. The syntax is as follows:

```
goto <state_label>;
```

Specifying the Trigger Position

The SignalTap II Logic Analyzer allows you to specify the amount of data that is acquired before and after a trigger event. You can set the trigger position independently between a Runtime and Power-Up Trigger. Select the desired ratio of pre-trigger data to post-trigger data by choosing one of the following ratios:

- **Pre—**This selection saves signal activity that occurred after the trigger (12% pre-trigger, 88% post-trigger).
- **Center—**This selection saves 50% pre-trigger and 50% post-trigger data.
- **Post—**This selection saves signal activity that occurred before the trigger (88% pre-trigger, 12% post-trigger).

These pre-defined ratios apply to both circular buffers and segmented buffers.

If you use the custom-state based triggering flow, you can specify a custom trigger position. The `segment_trigger` and `trigger` actions accept a post-fill count argument. The post-fill count specifies the number of samples to capture before stopping data acquisition for the circular buffer or a data segment when using the `trigger` and `segment_trigger` commands, respectively. When the captured data is displayed in the SignalTap II data window, the trigger position appears as the number of post-count samples from the end of the acquisition segment or buffer. Refer to Equation 1:

$$\text{Sample Number of Trigger Position} = (N - \text{Post-Fill Count}) \quad (1)$$

In this case, N is the sample depth of either the acquisition segment or circular buffer.

For segmented buffers, the acquisition segments that have a post-count argument defined use the post-count setting. Segments that do not have a post-count setting default to the trigger position ratios defined in the **Setup** tab.

For more details about the Custom-State based triggering flow, refer to "Custom State-Based Triggering" on page 13–36.

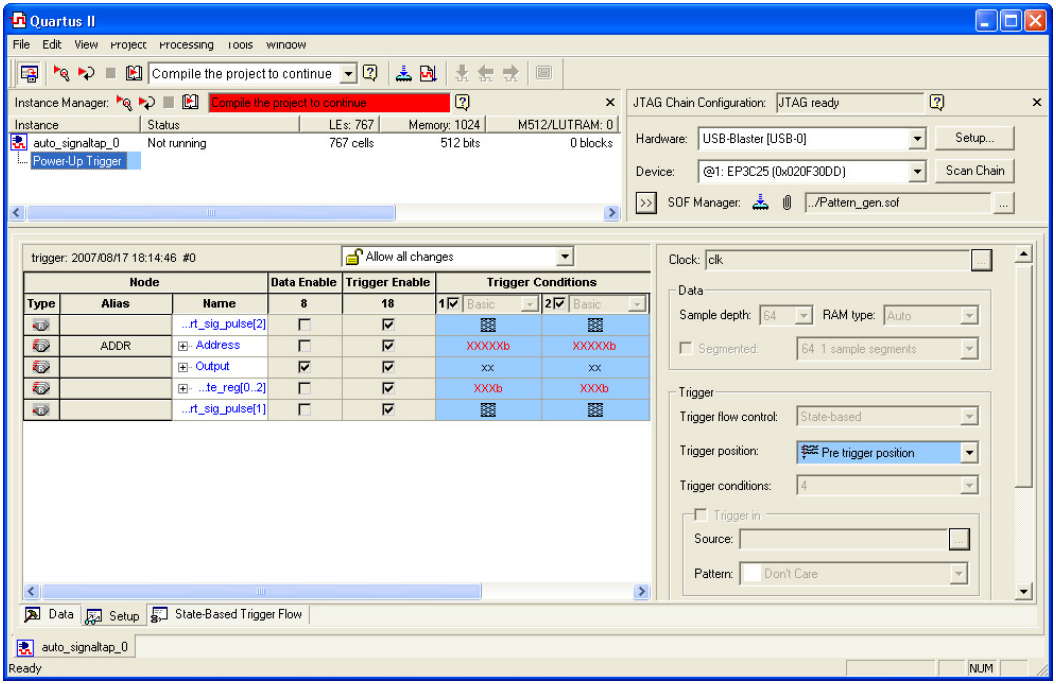
Creating a Power-Up Trigger

Typically, the SignalTap II Logic Analyzer is used to trigger on events that occur during normal device operation. You start an analysis manually once the target device is fully powered on and the device's JTAG connection is available. However, there may be cases when you would like to capture trigger events that occur during device initialization immediately after the FPGA is powered on or reset. With the SignalTap II Power-Up Trigger feature, you can capture data from triggers that occur after device programming but before the logic analyzer is started manually.

Enabling a Power-Up Trigger

A different Power-Up Trigger can be added to each logic analyzer instance in the SignalTap II Instance Manager. To enable the Power-Up Trigger for a logic analyzer instance, right-click the instance, and click **Enable Power-Up Trigger**, or select the instance, and on the Edit menu, click **Enable Power-Up Trigger**. To disable a Power-Up Trigger, click **Disable Power-Up Trigger** in the same locations. **Power-Up Trigger** is shown as a child instance below the name of the selected instance with the default trigger conditions set in the node list. Figure 13–20 shows the SignalTap II Editor when a Power-Up Trigger is enabled.

Figure 13–20. SignalTap II Editor with Power-Up Trigger Enabled



Managing and Configuring Power-Up and Runtime Trigger Conditions

When the Power-Up Trigger is enabled for a logic analyzer instance, you create basic and advanced trigger conditions for it in the same way you do with the regular trigger, also called the Runtime Trigger. Power-Up Trigger conditions that you can adjust are color coded light blue, while Run-Time Trigger conditions remain white. Since each instance now has two sets of trigger conditions, the Power-Up Trigger and the Run-Time Trigger, you can differentiate between the two with the color coding. To switch between the trigger conditions of the Power-Up Trigger and the Runtime Trigger, double-click the instance name or the Power-Up Trigger name in the Instance Manager.

You cannot make changes to the Power-Up Trigger conditions that would normally require a full recompile with Runtime Trigger conditions, such as adding signals, deleting signals, or changing between basic and advanced triggers. For these changes to be applied to the Power-up Trigger conditions, you must first make the changes using the Runtime Trigger conditions.



Any change made to the Power-Up Trigger conditions requires that the SignalTap II Logic Analyzer be recompiled, even if a similar change to the Runtime Trigger conditions does not require a recompilation.

While creating or making changes to the trigger conditions for the Run-Time Trigger or the Power-Up Trigger, you may want to copy these conditions to the other trigger. This makes it easy to look for the same trigger during both power-up and runtime. To do this, right-click the instance name or the Power-Up Trigger name in the Instance Manager, and click **Duplicate Trigger**, or select the instance name or the Power-Up Trigger name and, on the Edit menu, click **Duplicate Trigger**.

For information about running the SignalTap II Logic Analyzer instance with a Power-Up Trigger enabled, refer to "Running with a Power-Up Trigger" on page 13–60.

Using External Triggers

You can create a trigger input that allows you to trigger the SignalTap II Logic Analyzer from an external source. The external trigger input behaves like trigger condition `1`. It is evaluated and must be true before any other configured trigger conditions are evaluated. The analyzer can also supply a signal to trigger external devices or other SignalTap II instances. These features allow you to synchronize external logic analysis equipment with the internal logic analyzer. Power-Up Triggers can use the external triggers feature, but they must use the same source or target signal as their associated Run-Time Trigger.

Trigger In

To use Trigger In, perform the following steps:

1. In the SignalTap II Editor, click the **Setup** tab.
2. If a Power-Up Trigger is enabled, make sure you are viewing the Runtime Trigger conditions.
3. In the **Signal Configuration** pane, turn on **Trigger In**.
4. In the **Pattern** list, select the condition you want to act as your trigger event. You can set this separately for a Runtime or a Power-Up Trigger.
5. Click **Browse** next to the **Source** field in the Trigger In pane (Figure 13–22 on page 13–50). The **Node Finder** dialog box appears.

6. In the **Node Finder** dialog box, select the signal (either an input pin or an internal signal) that you want to drive the Trigger In source, and click **OK**.

   If you type a new signal name in the **Source** field, you create a new node that you can assign to an input pin in the Pin Planner or Assignment editor. If you leave the **Source** field blank, a default name is entered in the form `auto_stp_trigger_in_<`*SignalTap instance number*`>`.

Trigger Out

To use Trigger Out, perform the following steps:

1. In the SignalTap II Editor, click the **Setup** tab.

2. If a Power-Up trigger is enabled, make sure you are viewing the Runtime Trigger conditions.

3. In the **Signal Configuration** pane, turn on **Trigger Out** (refer to Figure 13–21 on page 13–49)

4. In the **Level** list, select the condition you want to signify that the trigger event is occurring. You can set this separately for a Run-Time or a Power-Up Trigger.

5. Type a new signal name in the **Target** field. A new node name is created that you must assign to an output pin in the Pin Planner or Assignment editor.

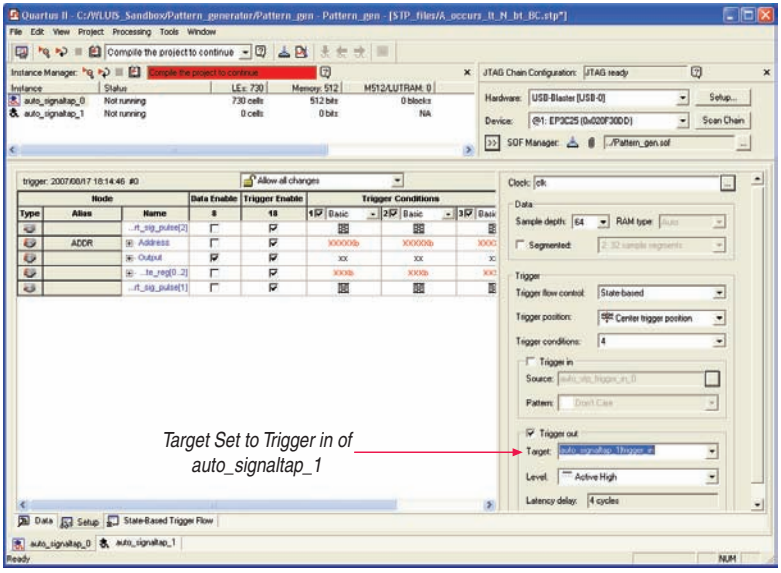
   If you leave the **Target** field blank, a default name is entered in the form `auto_stp_trigger_out_<`*SignalTap instance number*`>`. When the logic analyzer triggers, a signal at the level you indicated will be output on the pin you assigned to the new node.

Using the Trigger Out of One Analyzer as the Trigger In of Another Analyzer

An advanced feature of the SignalTap II Logic Analyzer is the ability to use the Trigger Out of one analyzer as the Trigger In to another analyzer. This feature allows you to synchronize and debug events that occur across multiple clock domains.

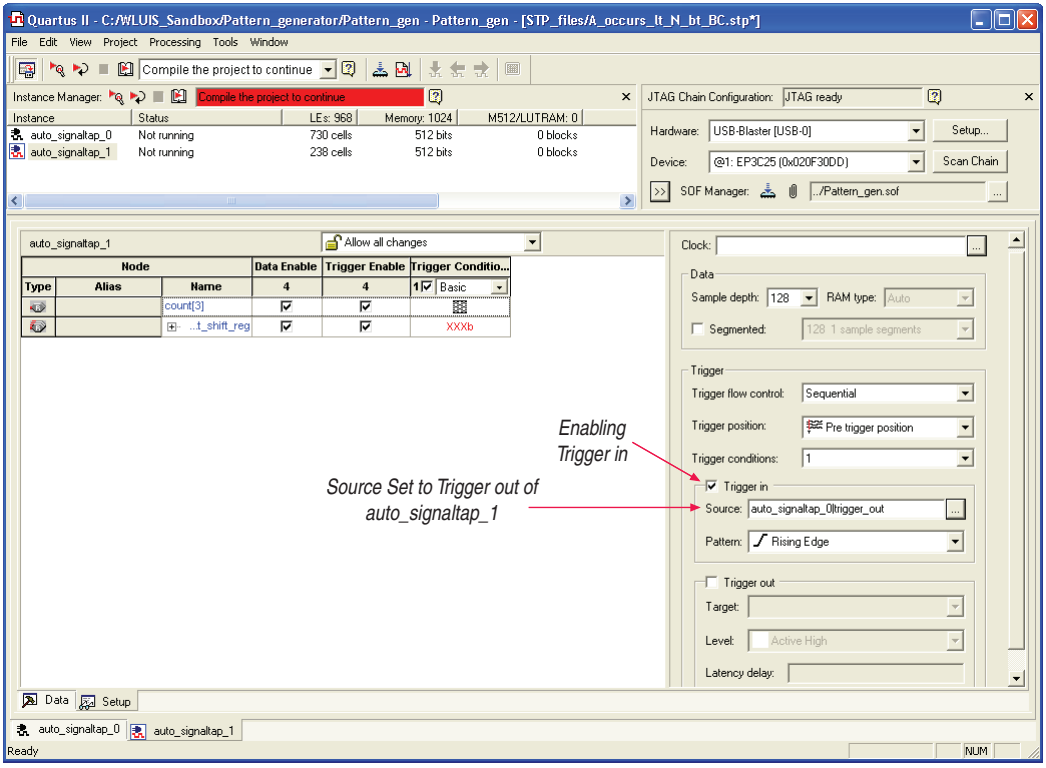
To perform this operation, first enable the **Trigger Out** of the source logic analyzer instance. On the Trigger out **Target** list, select the targeted logic analyzer instance. For example, if the instance named `auto_signaltap_0` should trigger `auto_signaltap_1`, select `auto_signaltap_1|trigger_in` from the list (Figure 13–21).

Figure 13–21. Configuring the Trigger Out Signal



- This automatically enables the Trigger In of the targeted logic analyzer instance and fills in the Trigger In **Source** field with the Trigger Out signal from the source logic analyzer instance. In this example, `auto_signaltap_0` is targeting `auto_signaltap_1`. The Trigger In **Source** field of `auto_signaltap_1` is automatically filled in with `auto_signaltap_0|trigger_out` (Figure 13–22).

Figure 13–22. Configuring the Trigger In Signal



Compile the Design

When you add a SignalTap II file to your project, the SignalTap II Logic Analyzer becomes part of your design. You must compile your project to incorporate the SignalTap II logic and enable the JTAG connection that is used to control the logic analyzer. When you are debugging with a traditional external logic analyzer, it is often necessary to make changes to the signals monitored as well as the trigger conditions. Since these adjustments often translate into recompilation time when using the SignalTap II Logic Analyzer, you can use the SignalTap II Logic Analyzer feature along with incremental compilation in the Quartus II software to reduce time spent recompiling.

Faster Compilations with Quartus II Incremental Compilation

To use Incremental compilation with the SignalTap II Logic Analyzer, you must perform the following steps:

- Enable Full Incremental Compilation for your design
- Assign design partitions
- Set partitions to the proper preservation levels
- Enable SignalTap for your design
- Add signals to SignalTap using the appropriate netlist filter in the node finder (either SignalTap II: pre-synthesis or SignalTap II: post-fitting).

When you compile your design with a SignalTap II file, the `sld_signaltap` and `sld_hub` entities are automatically added to the compilation hierarchy. These two entities are the main components of the SignalTap II Logic Analyzer, providing the trigger logic and JTAG interface required for operation.

Incremental compilation enables you to preserve the synthesis and fitting results of your original design and add the SignalTap II Logic Analyzer to your design without recompiling your original source code. This feature is also useful when you want to modify the configuration of the SignalTap II file. For example, you can modify the buffer sample depth or memory type without performing a full compilation after the change is made. Only the SignalTap II Logic Analyzer, configured as its own design partition, must be recompiled to reflect the changes.

To use incremental compilation, you must first enable **Full Incremental Compilation** for your design if it is not already enabled, assign design partitions if necessary, and set the design partitions to the correct preservation levels. Incremental compilation is the default setting for new projects in the Quartus II software, so you can establish design partitions immediately in a new project. However, it is not necessary to create any design partitions to use the SignalTap II Incremental Compilation feature. Once your design is set up to use full incremental compilation, the SignalTap II Logic Analyzer acts as its own separate design partition. You can begin taking advantage of incremental compilation by using the **SignalTap II: post-fitting filter** in the **Node Finder** to add signals for logic analysis.

Enabling Incremental Compilation for your Design

To enable Incremental Compilation if it is not already enabled, perform the following steps:

1. On the Assignments menu, click **Design Partitions window**.

2. In the **Incremental Compilation** list, select **Full Incremental Compilation**.

3. Create user-defined partitions if desired and set the **Netlist Type** to **Post-fit** for all partitions.



The netlist type for the top-level partition defaults to source. To take advantage of incremental compilation, you must set the Netlist types for the partitions you wish to tap as post-fit.

4. On the Processing menu, click **Start Compilation,** or click **Start Compilation** on the toolbar.

Your project is fully compiled the first time, establishing the design partitions you have created. When enabled for your design, the SignalTap II Logic Analyzer will always be a separate partition. After the first compilation, you can use the SignalTap II Logic Analyzer to analyze signals from the post-fit netlist. If your partitions are set correctly, subsequent compilations due to SignalTap II settings are able to take advantage of the shorter compilation times.



For more information about configuring and performing Incremental Compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Using Incremental Compilation with the SignalTap II Logic Analyzer

The SignalTap II Logic Analyzer is automatically configured to work with the incremental compilation flow. For all signals that you want to connect to the SignalTap II Logic Analyzer from the post-fit netlist, set the netlist type of the partition containing the desired signals to Post-Fit or Post-Fit (Strict) with a Fitter Preservation Level of Placement and Routing using the Design Partitions window. Use the **SignalTap II: post-fitting filter** in the **Node Finder** to add the signals of interest to your SignalTap II configuration file. If you want to add signals from the pre-synthesis netlist, set the netlist type to Source File and use the **SignalTap II: pre-synthesis filter** in the **Node Finder**. Do not use the netlist type **Post-Synthesis** with the SignalTap II Logic Analyzer.



Be sure to conform to the following guidelines when using post-fit/pre-synthesis nodes:

- Read all incremental compilation guidelines to ensure the proper partition of a project.
- To speed compile time, use only post-fit nodes for partitions set to preservation level post-fit.

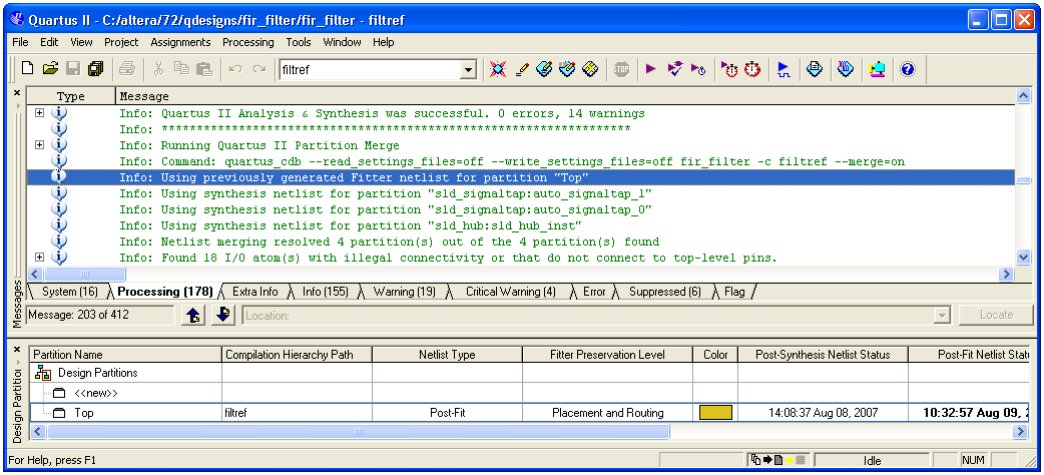
- Do not mix pre-synthesis and post-fit nodes in any partition. If you must tap presynthesis nodes for a particular partition, make all tapped nodes in that partition presynthesis nodes and change the netlist type to **source** in the design partitions window.

Node names may be different between a pre-synthesis netlist and a post-fit netlist. In general, registers and user input signals share common names between the two netlists. During compilation, certain optimizations will change the names of combinational signals in your RTL. If the type of node name chosen does not match the netlist type, the compiler may not be able to find the signal to connect to your SignalTap II Logic Analyzer instance for analysis. The compiler will issue a critical warning to warn you of this scenario. The signal that is not connected is tied to ground in the SignalTap II data tab.

If you do use incremental compile flow with the SignalTap II Logic Analyzer and source file changes are necessary, be aware that you may have to remove compiler-generated post-fit net names. Source code changes force the affected partition to go through a resynthesis. During synthesis, the compiler cannot find compiler-generated net names from a previous compilation. Altera recommends you use only registered and user input signals as debugging taps in your STP file whenever possible. Both registered and user-supplied input signals share common node names in the pre-synthesis and post-fit netlist. As a result, using only registered and user-supplied input signals in your STP file limits the changes you need to make to your SignalTap configuration.

To verify that your original design was not modified, examine the messages in the **Partition Merge** section of the Compilation Report. Figure 13–23 shows an example of the messages displayed.

Figure 13–23. Compilation Report Messages



Unless you make changes to your design partitions that require recompilation, only the SignalTap II design partition is recompiled. If you make subsequent changes to only the SignalTap II file, only the SignalTap II design partition must be recompiled, reducing your recompilation time.

Preventing Changes Requiring Recompilation

You can configure the SignalTap II file to prevent changes that normally require a recompilation. You do this by selecting a lock mode from above the node list in the **Setup** tab. Whether or not you are using incremental compilation, you can lock your configuration by choosing to allow only trigger condition changes.

For more information about the use of lock modes, refer to the Quartus II Help.

Timing Preservation with the SignalTap II Logic Analyzer

In addition to verifying functionality, timing closure is one of the most crucial processes in successfully completing a design. When you compile a project with a SignalTap II Logic Analyzer without the use of incremental compilation, you add IP to your existing design. Therefore, you can affect the existing placement, routing, and timing of your design. To minimize the effect that the SignalTap II Logic Analyzer has on your design, Altera recommends that you use incremental compilation for

your project. Incremental compilation is the default setting in new designs and can be easily enabled and configured in existing designs. With the SignalTap II Logic Analyzer in its own design partition, it has little to no affect on your design.

In addition to using the incremental compilation flow for your design, you can use the following techniques to help maintain timing:

- Avoid adding critical path signals to your SignalTap II file.
- Minimize the number of combinational signals you add to your SignalTap II file, and add registers whenever possible.
- Specify an f_{MAX} constraint for each clock in your design.



For an example of timing preservation with the SignalTap II Logic Analyzer, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Performance and Resource Considerations

There is an inherent trade-off between runtime flexibility of the SignalTap II Logic Analyzer, timing performance of the Signal Tap II Logic Analyzer, and the resource usage. The SignalTap II Logic Analyzer allows you to select the runtime configurable parameters to balance the need for runtime flexibility, speed, and area. The default values have been chosen to provide maximum flexibility so you can reach debugging closure as quickly as possible; however, you can adjust these settings to determine whether there is a more optimal configuration for your design.

The suggestions in this section provide some tips to provide extra timing slack if you have determined that the SignalTap II logic is in your critical path, or to alleviate the resource requirements that the SignalTap II Logic Analyzer consumes if your design is resource-constrained.

If the SignalTap II logic is part of your critical path, the following suggestions can help to speed up the performance of the SignalTap II Logic Analyzer:

- **Disable runtime configurable options**—Certain resources are allocated to accommodate for run-time flexibility. If you are using either advanced triggers or the state-based triggering flow, you can disable run-time configurable parameters for a boost in f_{MAX} of the SignalTap II logic. If you are using the state-based triggering flow, try disabling the **Goto state destination** option and performing a recompilation before disabling the other runtime configurable options. The **Goto state destination** option has the greatest impact on f_{MAX} as compared to the other runtime configurable options.

- **Minimize the number of signals that have Trigger Enable selected**—All of the signals that you add to the SignalTap II file have Trigger Enable turned on. Turn off Trigger Enable for signals that you do not plan to use as triggers.

- **Turn on Physical Synthesis for register retiming**—If you have a large number of triggering signals enabled (greater than the number of inputs that would fit in a LAB) that fan-in to logic gate-based triggering condition, such as a basic trigger condition or a logical reduction operator in the advanced trigger tab, turn on the **Perform register retiming**. This can help balance combinational logic across LABs.

If your design is resource constrained, the following suggestions can help to reduce the amount of logic or memory used by the SignalTap II Logic Analyzer:

- **Disable runtime configurable options**—Disabling runtime configurability for the advanced trigger conditions or the runtime configurable options in the state-based triggering flow will result in less LE usage.

- **Minimize the number of segments in the acquisition buffer**—You can reduce the number of logic resources used for the SignalTap II Logic Analyzer by limiting the number of segments in your sampling buffer to only that which is required.

- **Disable the Data Enable for signals that are used for triggering only**—By default, both the data enable and trigger enable options are selected for all signals. Turning off the data enable option for signals used as trigger inputs only will save on memory resources used by the SignalTap II Logic Analyzer.

Because performance results are design-dependent, try these options in different combinations until you achieve the desired balance between functionality, performance, and utilization.



For more information about area and timing optimization, refer the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Program the Target Device or Devices

Once your project, including the SignalTap II Logic Analyzer, is compiled, you must configure the FPGA target device. When you are using the SignalTap II Logic Analyzer for debugging, you can configure the device from the SignalTap II file instead of the Quartus II Programmer. Because you configure from the SignalTap II file, you can open more than one SignalTap II file and program multiple devices to debug multiple designs simultaneously.

The settings in a SignalTap II file must be compatible with the programming (SOF) file used to program the device. A SignalTap II file is considered compatible with an SOF when the settings for the logic analyzer, such as the size of the capture buffer and the signals selected for monitoring or triggering, match the way the target device will be programmed. If the files are not compatible, you will still be able to program the device, but you will not be able to run or control the logic analyzer from the SignalTap II Editor.

To ensure programming compatibility, make sure to program your device with the latest SOF created from the most recent compilation.

Before starting a debugging session, do not make any changes to the SignalTap II file settings that would require the project to be recompiled. You can check the SignalTap II status display at the top of the Instance Manager to see if a change you made requires the design to be recompiled, producing a new SOF. This gives you the opportunity to undo the change, so that a recompilation is not necessary. To prevent any such changes, enable a lock mode in the SignalTap II file.

Programming a Single Device

To configure a single device for use with the SignalTap II Logic Analyzer, perform the follow steps:

1. In the **JTAG Chain Configuration** pane in the SignalTap II Editor, select the connection you use to communicate with the device from the **Hardware** list. If you need to add your communication cable to the list, click **Setup** to configure your connection.

2. Click **Browse** in the **JTAG Chain Configuration** pane, and select the SOF file that includes the compatible SignalTap II Logic Analyzer.

3. Click **Scan Chain**. The Scan Chain operation enumerates all of the JTAG devices within your JTAG chain.

4. In the **Device** list, select the device to which you want to download the design. The device list shows an ordered list of all devices in the JTAG chain.

   All of the devices are numbered sequentially according to their position in the JTAG chain, prefixed with the "@". For example: `@1 : EP3C25 (0x020F30DD)` lists a Cyclone III device as the first device in the chain, with the JTAG ID code of `0x020F30DD`.

5. Click the **Program Device** icon.

Programming Multiple Devices to Debug Multiple Designs

You can simultaneously debug multiple designs using one instance of the Quartus II software by performing the following steps:

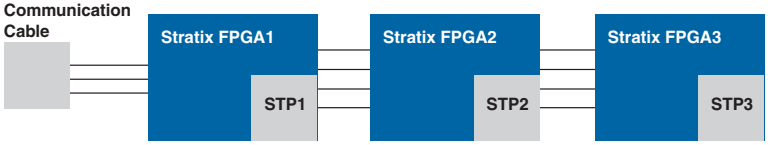
1. Create, configure, and compile each project that includes a SignalTap II file.
2. Open each SignalTap II file.

   You do not have to open a Quartus II project to open a SignalTap II file.

3. Use the **JTAG Chain Configuration** pane controls to select the target device in each SignalTap II file.
4. Program each FPGA.
5. Run each analyzer independently.

Figure 13–24 shows a JTAG chain and its associated SignalTap II files.

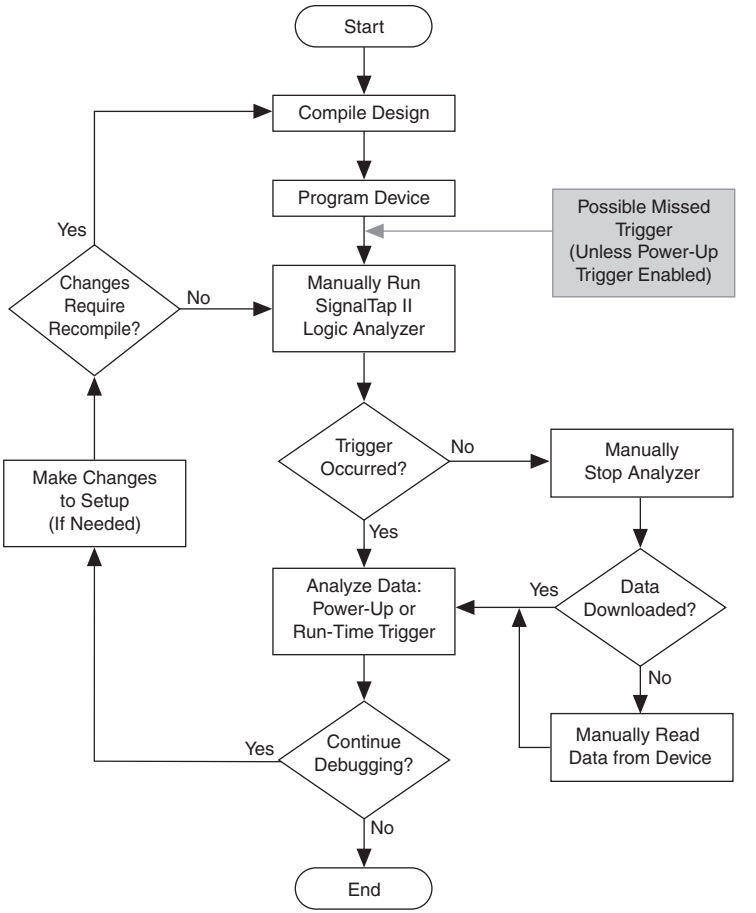
Figure 13–24. JTAG Chain



Run the SignalTap II Logic Analyzer

After the device is configured with your design that includes the SignalTap II Logic Analyzer, you can perform debugging operations in a manner similar to the use of an external logic analyzer. You "arm" the logic analyzer by starting an analysis. When your trigger event occurs, the captured data is stored in the memory buffer on the device and then transferred to the SignalTap II file over the JTAG connection. You can also perform the equivalent of a "force trigger" that lets you view the captured data currently in the buffer without a trigger event occurring. Figure 13–25 illustrates a flow that shows how you operate the SignalTap II Logic Analyzer. The flowchart indicates where Power-Up and Run-Time Trigger events occur and when captured data from these events is available for analysis.

Figure 13–25. Power-Up and runtime Trigger Events Flowchart



The SignalTap II toolbar in the Instance Manager has four options for running the analyzer:

- **Run Analysis**—The SignalTap II Logic Analyzer runs until the trigger event occurs. When the trigger event occurs, monitoring and data capture stops once the acquisition buffer is full.
- **AutoRun Analysis**—The SignalTap II Logic Analyzer continuously captures data until the **Stop Analysis** button is clicked, ignoring all trigger event conditions.
- **Stop Analysis**—SignalTap II analysis stops. The acquired data does not appear automatically if the trigger event has not occurred.
- **Read Data**—Captured data is displayed. This button is useful if you want to view the acquired data even if the trigger has not occurred.

Running with a Power-Up Trigger

If you have enabled and set up a Power-Up Trigger for an instance of the SignalTap II Logic Analyzer, the captured data may already be available for viewing if the trigger event occurred after device configuration. To download the captured data or to check if the Power-Up Trigger is still running, click **Run Analysis** in the Instance Manager. If the Power-Up Trigger occurred, the logic analyzer immediately stops, and the captured data is downloaded from the device. The data can now be viewed on the **Data** tab of the SignalTap II Editor. If the Power-Up Trigger did not occur, no captured data is downloaded, and the logic analyzer continues to run. You can wait for the Power-Up Trigger event to occur, or, to stop the logic analyzer, click **Stop Analysis**.

Running with Runtime Triggers

You can arm and run the SignalTap II Logic Analyzer manually after device configuration to capture data samples based on the Runtime Trigger. You can do this immediately if there is no Power-Up Trigger enabled. If a Power-Up Trigger is enabled, you can do this after the Power-Up Trigger data is downloaded from the device or once the logic analyzer is stopped because the Power-Up Trigger event did not occur. Click **Run Analysis** in the SignalTap II Editor to start monitoring for the trigger event. You can start multiple SignalTap II instances at the same time by selecting all of the required instances before you click **Run Analysis** on the toolbar.

Unless the logic analyzer is stopped manually, data capture begins when the trigger event evaluates to `TRUE`. When this happens, the captured data is downloaded from the buffer. You can view the data in the **Data** tab of the SignalTap II Editor.

Performing a Force Trigger

Sometimes when you use an external logic analyzer or oscilloscope, you want to see the current state of signals without setting up or waiting for a trigger event to occur. This is referred to as a "force trigger" operation, because you are forcing the test equipment to capture data without regard to any set trigger conditions. With the SignalTap II Logic Analyzer, you can choose to run the analyzer and capture data immediately or run the analyzer and capture data when you want.



For more information, refer to the *Design Debugging Using In-System Sources and Probes* chapter in volume 3 of the *Quartus II Handbook*.

To run the analyzer and immediately capture data, disable the trigger conditions by turning off each **Trigger Condition** column in the node list. This operation does not require a recompilation. Click **Run Analysis** in the Instance Manager. The SignalTap II Logic Analyzer immediately triggers, captures, and downloads the data to the **Data** tab of the SignalTap II Editor. If the data does not download automatically, click **Read Data** in the Instance Manager.

If you want to choose when to capture data manually, it is not required that you disable the trigger conditions. Click **Autorun Analysis** to start the logic analyzer, and click **Stop Analysis** to capture data. If the data does not download to the **Data** tab of the SignalTap II Editor automatically, click **Read Data**.

Finally, you can choose to capture data manually after a trigger event has occurred. This is useful if you still want the trigger event to occur, but you want to capture data about the signals at some point after the trigger without capturing the trigger event itself. To do this, set the **Buffer acquisition mode** to **Circular** and **Continuous**, and click **Run Analysis**. When the trigger event occurs, the status in the SignalTap II Health Monitor is shown as `Acquiring post-trigger data`, but the logic analyzer does not stop. When you want to capture and download the data, click **Stop Analysis**. If the data does not download automatically, click **Read Data**.



You can also use In-System Sources and Probes in conjunction with the SignalTap II Logic Analyzer to force trigger conditions. The In-System Sources and Probes feature allows you to drive and sample values on to selected nets over the JTAG chain. For more information, refer to the *Design Debugging Using In-System Sources and Probes* chapter in volume 3 of the *Quartus II Handbook*.

SignalTap II Status Messages

Table 13–11 describes the text messages that may appear in the SignalTap II Health Monitor in the Instance Manager before, during, and after a data acquisition. Use these messages to know the state of the logic analyzer or what operation it is performing.

Table 13–11. Text Messages in the SignalTap II Health Monitor

Message	Message Description
`Not running`	The SignalTap II Logic Analyzer is not running. There is no connection to a device or the device is not configured.
`(Power-Up Trigger) Waiting for clock` *(1)*	The SignalTap II Logic Analyzer is performing a Runtime or Power-Up Trigger acquisition and is waiting for the clock signal to transition.
`Acquiring (Power-Up) pre-trigger data` *(1)*	The trigger condition has not been evaluated yet. A full buffer of data is collected if the circular buffer acquisition mode is selected.
`Trigger In conditions met`	Trigger In condition has occurred. The SignalTap II Logic Analyzer is waiting for the condition of the first trigger condition to occur. This can appear if Trigger In is specified.
`Waiting for (Power-up) trigger` *(1)*	The SignalTap II Logic Analyzer is now waiting for the trigger event to occur.
`Trigger level <x> met`	The condition of trigger condition *x* has occurred. The SignalTap II Logic Analyzer is waiting for the condition specified in condition *x + 1* to occur.
`Acquiring (power-up) post-trigger data` *(1)*	The whole trigger event has occurred. The SignalTap II Logic Analyzer is acquiring the post-trigger data. The amount of post-trigger data collected is user-defined between 12%, 50%, and 88% when the circular buffer acquisition mode is selected.
`Offload acquired (Power-Up) data` *(1)*	Data is being transmitted to the Quartus II software through the JTAG chain.
`Ready to acquire`	The SignalTap II Logic Analyzer is waiting for the user to arm the analyzer.

Note to Table 13–11:

(1) This message can appear for both Runtime and Power-Up Trigger events. When referring to a Power-Up Trigger, the text in parentheses is added.



In segmented acquisition mode, pre-trigger and post-trigger do not apply.

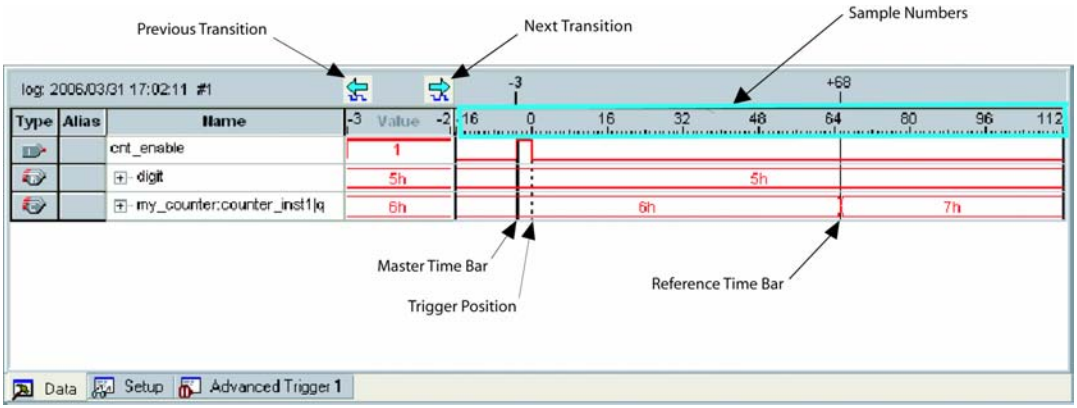
View, Analyze, and Use Captured Data

Once a trigger event has occurred or you capture data manually, you can use the SignalTap II interface to examine the data, and use your findings to help debug your design. The SignalTap II Logic Analyzer provides a number of features that makes it easy to do this.

Viewing Captured Data

You can view captured SignalTap II data in the **Data** tab of the SignalTap II file (Figure 13–26). Each row of the **Data** tab displays the captured data for one signal or bus. Buses can be expanded to show the data for each individual signal on the bus. Click on the data waveforms to zoom in on the captured data samples, and right-click to zoom out.

Figure 13–26. Captured SignalTap II Data



When you are viewing captured data, it is often useful to know the time interval between two events. Time bars enable you to see the number of clock cycles between two samples of captured data in your system. There are two types of time bars:

- **Master Time Bar**—The master time bar's label displays the absolute time of its location in bold. The master time bar is a thick black line in the **Data** tab. The captured data has only one master time bar.
- **Reference Time Bar**—The reference time bar's label displays the time relative to the master time bar. You can create an unlimited number of reference time bars.

To help you find a transition of signals relative to the master time bar location, use either the **Next Transition** or the **Previous Transition** button. This aligns the master time bar with the next or previous

transition of a selected signal or group of selected signals. This feature is very useful when the sample depth is very large and the rate at which signals toggle is very low.

Creating Mnemonics for Bit Patterns

The mnemonic table feature allows you to assign a meaningful name to a set of bit patterns, such as a bus. To create a mnemonic table, right-click in the **Setup** or **Data** tab of a SignalTap II file, and click **Mnemonic Table Setup**. You create a mnemonic table by entering sets of bit patterns and specifying a label to represent each pattern. Once you have created a mnemonic table, you assign it to a group of signals. To assign a mnemonic table, right-click on the group, click **Bus Display Format**, and select the desired mnemonic table.

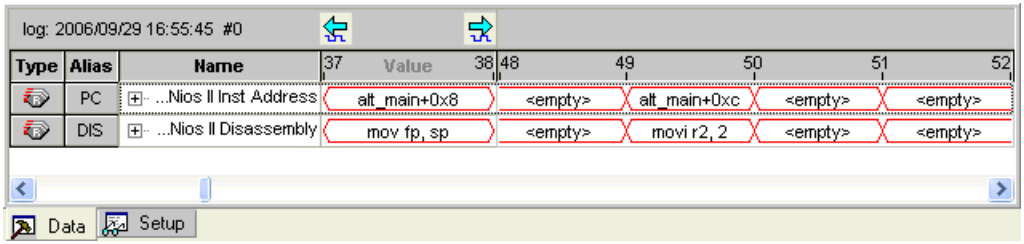
The labels you create in a table are used in different ways on the **Setup** and **Data** tabs. On the **Setup** tab, you can create basic triggers with meaningful names by right-clicking an entry in any Trigger Conditions column and selecting a label from the table you assigned to the signal group. On the **Data** tab, if any captured data matches a bit pattern contained in an assigned mnemonic table, the signal group data is replaced with the appropriate label, making it easy to see when expected data patterns occur.

Automatic Mnemonics with a Plug-In

When you use a plug-in to add signals to a SignalTap II file, mnemonic tables for the added signals are automatically created and assigned to the signals defined in the plug-in. If you ever need to manually enable these mnemonic tables, right-click on the name of the signal or signal group. On the **Bus Display Format** submenu, click the name of the mnemonic table that matches the plug-in.

As an example, the Nios II plug-in makes it easy to monitor your design's signal activity as code is executed. If you have set up the logic analyzer to trigger on a function name in your Nios II code based on data from an ELF file, you can see the function name in the Instance Address signal group at the trigger sample, along with the corresponding disassembled code in the Disassembly signal group, as shown in Figure 13–27 on page 13–65. Captured data samples around the trigger are referenced as offset addresses from the trigger function name.

Figure 13–27. Data Tab when the Nios II Plug-In is Used



Locating a Node in the Design

When you find the source of a bug in your design using the SignalTap II Logic Analyzer, you can use the node locate feature to locate that signal in many of the tools found in the Quartus II software, as well as in your design files. This lets you find the source of the problem quickly so you can modify your design to correct the flaw. To locate a signal from the SignalTap II Logic Analyzer in one of the Quartus II software tools or your design files, right-click on the signal in the SignalTap II file, and click **Locate in** <*tool name*>. You can locate a signal from the node list in any of the following locations:

- Assignment Editor
- Pin Planner
- Timing Closure Floorplan
- Chip Planner
- Resource Property Editor
- Technology Map Viewer
- RTL Viewer
- Design File

For more information about using these tools, refer to the appropriate chapters in the *Quartus II Handbook*.

Saving Captured Data

The data log shows the history of captured data and the triggers used to capture the data. The analyzer acquires data, stores it in a log, and displays it as waveforms. When the logic analyzer is in auto-run mode and a trigger event occurs more than once, captured data for each time the trigger occurred is stored as a separate entry in the data log. This makes it easy to go back and review the captured data for each trigger event. The default name for a log is based on the time when the data was acquired. Altera recommends that you rename the data log with a more meaningful name.

The logs are organized in a hierarchical manner; similar logs of captured data are grouped together in trigger sets. If the Data Log pane is closed, on the View menu, select **Data Log** to reopen it. To enable data logging, turn on **Enable data log** in the **Data Log** (Figure 13–11). To recall a data log for a given trigger set and make it active, double-click the name of the data log in the list.

The Data Log feature is useful for organizing different sets of trigger conditions and different sets of signal configurations. Refer to "Managing Multiple SignalTap II Files and Configurations" on page 13–28.

Converting Captured Data to Other File Formats

You can export captured data in the following file formats, some of which can be used with other EDA simulation tools:

- Comma Separated Values File (**.csv**)
- Table File (**.tbl**)
- Value Change Dump File (**.vcd**)
- Vector Waveform File (**.vwf**)
- Graphics format files (**.jpg**, **.bmp**)

To export the SignalTap II Logic Analyzer's captured data, on the File menu, click **Export** and specify the **File Name**, the **Export Format**, and the **Clock Period**.

Creating a SignalTap II List File

Captured data can also be viewed in a SignalTap II list file. A SignalTap II list file is a text file that lists all the data captured by the logic analyzer for a trigger event. Each row of the list file corresponds to one captured sample in the buffer. Columns correspond to the value of each of the captured signals or signal groups for that sample. If a mnemonic table was created for the captured data, the numerical values in the list are replaced with a matching entry from the table. This is especially useful with the use of a plug-in that includes instruction code disassembly. You can immediately see the order in which the instruction code was executed during the same time period of the trigger event. To create a SignalTap II list file, on the File menu, select **Create/Update,** and click **Create SignalTap II List File**.

Other Features

The SignalTap II Logic Analyzer has a number of other features that do not necessarily belong to a particular task in the task flow.

Using the SignalTap II MATLAB MEX Function to Capture Data

If you use MATLAB for DSP design, you can call the MATLAB MEX function `alt_signaltap_run`, built into the Quartus II software, to acquire data from the SignalTap II Logic Analyzer directly into a matrix in the MATLAB environment. If you use the MEX function repeatedly in a loop, you can perform as many acquisitions as you can when using SignalTap II in the Quartus II software environment in the same amount of time.



The SignalTap II MATLAB MEX function is available only in the Windows version of the Quartus II software. It is compatible with MATLAB Release 14 Original Release Version 7 and later.

To set up the Quartus II software and the MATLAB environment to perform SignalTap II acquisitions, perform the following steps:

1. In the Quartus II software, create a SignalTap II file.
2. In the node list in the **Data** tab of the SignalTap II Editor, organize the signals and groups of signals into the order in which you want them to appear in the MATLAB matrix. Each column of the imported matrix represents a single SignalTap II acquisition sample, while each row represents a signal or group of signals in the order they are organized in the **Data** tab.



Signal groups acquired from the SignalTap II Logic Analyzer and transferred into the MATLAB environment with the MEX function are limited to a width of 32 signals. If you want to use the MEX function with a bus or signal group that contains more than 32 signals, split the group up into smaller groups that do not exceed the 32 signal limit.

3. Save the SignalTap II file and compile your design. Program your device and run the SignalTap II Logic Analyzer to make sure your trigger conditions and signal acquisition are working correctly.

4. In the MATLAB environment, add the Quartus II binary directory to your path with the following command:

```
addpath <Quartus install directory>\win ↵
```

You can view the help file for the MEX function by entering `alt_signaltap_run` in MATLAB without any operators.

You use the MEX function in the MATLAB environment to open the JTAG connection to the device and run the SignalTap II Logic Analyzer to acquire data. When you finish acquiring data, you must close the connection.

To open the JTAG connection and begin acquiring captured data directly into a MATLAB matrix called `stp`, use the following command:

```
stp = alt_signaltap_run('<stp filename>'[,('signed'|'unsigned')[,'<instance names>'[,/
'<signalset name>'[,'<trigger name>']]]]); ↵
```

When capturing data, *<stp filename>* is the name of the SignalTap II file you want to use. This is required for using the MEX function. The other MEX function options are defined in Table 13–12.

Table 13–12. SignalTap II MATLAB MEX Function Options (Part 1 of 2)

Option	Usage	Description
`signed` `unsigned`	`'signed'` `'unsigned'`	The **signed** option turns signal group data into 32-bit two's complement signed integers. The most significant bit (MSB) of the group as defined in the SignalTap II **Data** tab is the sign bit. The **unsigned** option keeps the data as an unsigned integer. The default is **signed**.

Table 13–12. SignalTap II MATLAB MEX Function Options (Part 2 of 2)

Option	Usage	Description
<instance name>	'auto_signaltap_0'	Specify a SignalTap II instance if more than one instance is defined. The default is the first instance in the SignalTap II file, auto_signaltap_0.
<signal set name> *<trigger name>*	'my_signalset' 'my_trigger'	Specify the signal set and trigger from the SignalTap II data log if multiple configurations are present in the SignalTap II file. The default is the active signal set and trigger in the file.

You can enable or disable verbose mode to see the status of the logic analyzer while it is acquiring data. To enable or disable verbose mode, use the following commands:

```
alt_signaltap_run('VERBOSE_ON'); ↵
alt_signaltap_run('VERBOSE_OFF'); ↵
```

When you finish acquiring data, you must close the JTAG connection. Use the following command to close the connection:

```
alt_signaltap_run('END_CONNECTION'); ↵
```

For more information about the use of MEX functions in MATLAB, refer to the MATLAB Help.

Using SignalTap II in a Lab Environment

You can install a stand-alone version of the SignalTap II Logic Analyzer. This version is particularly useful in a lab environment where you do not have a workstation that meets the requirements for a complete Quartus II installation, or if you do not have a license for a full installation of the Quartus II software. The stand-alone version of the SignalTap II Logic Analyzer is included with the Quartus II stand-alone Programmer and is available from the Downloads page of the Altera website, **www.altera.com**.

Remote Debugging Using the SignalTap II Logic Analyzer

You can use the SignalTap II Logic Analyzer to debug a design that is running on a device attached to a PC in a remote location.

To perform a remote debugging session, you must have the following setup:

- The Quartus II software installed on the local PC
- Stand-alone SignalTap II Logic Analyzer or the full version of the Quartus II software installed on the remote PC
- Programming hardware connected to the device on the PCB at the remote location
- TCP/IP protocol connection

Equipment Setup

On the PC in the remote location, install the stand-alone version of the SignalTap II Logic Analyzer or the full version of the Quartus II software. This remote computer must have Altera programming hardware connected, such as the EthernetBlaster or USB-Blaster.

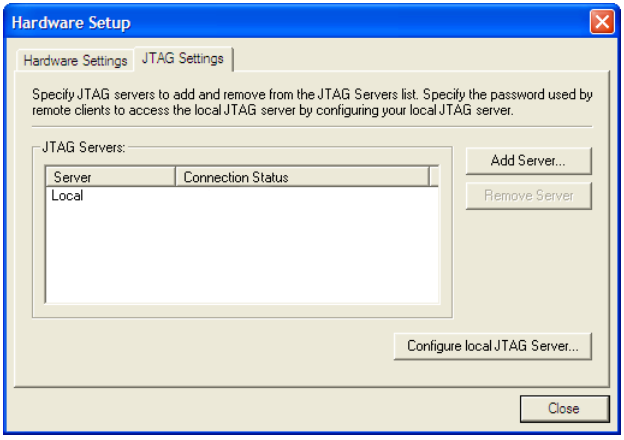
On the local PC, install the full version of the Quartus II software. This local PC must be connected to the remote PC across a LAN with the TCP/IP protocol.

Software Setup on the Remote PC

To setup the software on the remote PC, perform the following steps:

1. In the Quartus II programmer, click **Hardware Setup**.
2. Click the **JTAG Settings** tab (Figure 13–28 on page 13–70).

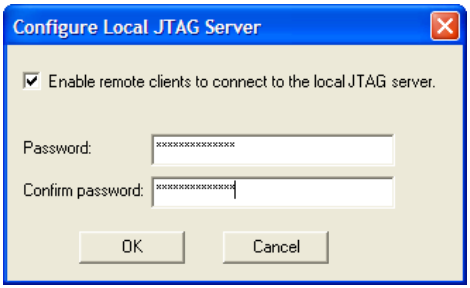
Figure 13–28. Configure JTAG on Remote PC



3. Click **Configure local JTAG Server**.

4. In the **Configure Local JTAG Server** dialog box (Figure 13–29), turn on **Enable remote clients to connect to the local JTAG server,** and type your password in the password box. Type your password again in the **Confirm Password** box and click **OK**.

Figure 13–29. Configure Local JTAG Server on Remote



Software Setup on the Local PC

To set up your software on your local PC, perform the following steps:

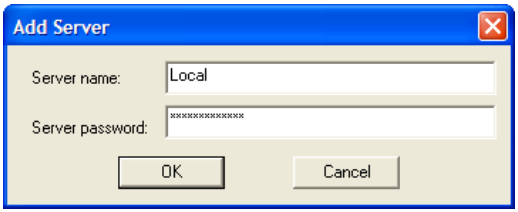
1. Launch the Quartus II programmer.

2. Click **Hardware Setup.**

3. On the **JTAG settings** tab, click **Add server**.

4. In the **Add Server** dialog box (Figure 13–30), type the network name or IP address of the server you want to use and the password for the JTAG server that you created on the remote PC.

Figure 13–30. Add Server Dialog Box



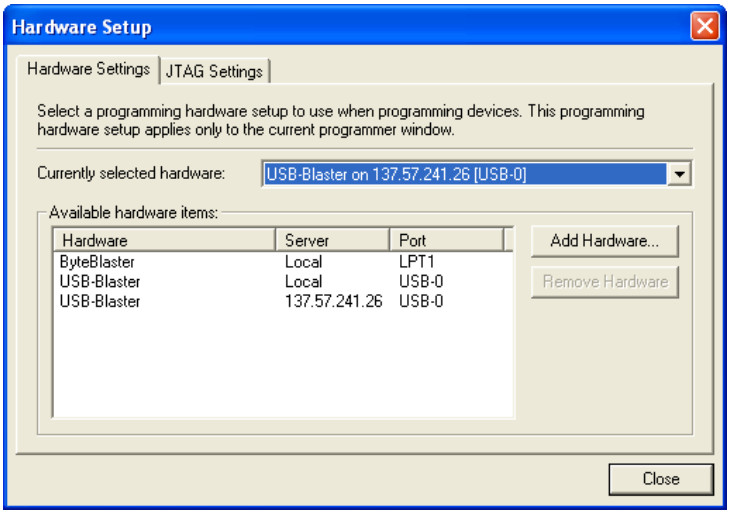
5. Click **OK**.

SignalTap II Setup on the Local PC

To connect to the hardware on the remote PC, perform the following steps:

1. Click the **Hardware Settings** tab and select the hardware on the remote PC (Figure 13–31).

Figure 13–31. Selecting Hardware on Remote PC



2. Click **Close**.

You can now control the logic analyzer on the device attached to the remote PC as if it was connected directly to the local PC.

SignalTap II Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF format.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

SignalTap II Command Line Options

To compile your design with the SignalTap II Logic Analyzer using the command prompt, you must use the `quartus_stp` command. Table 13–13 shows the options that help you better understand how to use the **quartus_stp** executable.

Table 13–13. SignalTap II Command-Line Options (Part 1 of 2)

Option	Usage	Description
`stp_file`	`quartus_stp --stp_file <stp_filename>`	Assigns the specified SignalTap II file to the `USE_SIGNALTAP_FILE` in the Quartus II Settings File (QSF).
`enable`	`quartus_stp --enable`	Creates assignments to the specified SignalTap II file in the QSF, and changes `ENABLE_SIGNALTAP` to `ON`. The SignalTap II Logic Analyzer is included in your design the next time the project is compiled. If no SignalTap II file is specified in the QSF, the `--stp_file` option must be used. If the `--enable` option is omitted, the current value of `ENABLE_SIGNALTAP` in the QSF is used.

Table 13–13. SignalTap II Command-Line Options (Part 2 of 2)

Option	Usage	Description
disable	quartus_stp --disable	Removes the SignalTap II file reference from the QSF and changes ENABLE_SIGNALTAP to OFF. The SignalTap II Logic Analyzer is removed from the design database the next time you compile your design. If the --disable option is omitted, the current value of ENABLE_SIGNALTAP in the QSF is used.
create_signaltap_hdl_file	quartus_stp -- create_signaltap_hdl_file	Creates a SignalTap II file representing the SignalTap II instance in the design generated by the SignalTap II Logic Analyzer megafunction created with the MegaWizard Plug-in Manager. The file is based on the last compilation. You must use the --stp_file option to properly create a SignalTap II file. Analogous to **Create SignalTap II File from Design Instance(s)** command in the Quartus II software.

Example 13–2 illustrates how to compile a design with the SignalTap II Logic Analyzer at the command line:

Example 13–2.

```
quartus_stp filtref --stp_file stp1.stp --enable ↵
quartus_map filtref --source=filtref.bdf --family=CYCLONE ↵
quartus_fit filtref --part=EP1C12Q240C6 --fmax=80MHz --tsu=8ns ↵
quartus_tan filtref ↵
quartus_asm filtref ↵
```

The quartus_stp --stp_file stp1.stp --enable command creates the QSF variable and instructs the Quartus II software to compile the **stp1.stp** file with your design.

Example 13–3 shows how to create a new SignalTap II file after building the SignalTap II Logic Analyzer instance with the MegaWizard Plug-In Manager:

Example 13–3.

```
quartus_stp filtref --create_signaltap_hdl_file --stp_file stp1.stp ↵
```



For information about the other command line executables and options refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

SignalTap II Tcl Commands

The **quartus_stp** executable supports a Tcl interface that allows you to capture data without running the Quartus II GUI. You cannot execute SignalTap II Tcl commands from within the Tcl console in the GUI. They must be run from the command line with the **quartus_stp** executable. To run a Tcl file that has SignalTap II Tcl commands, use the following command:

```
quartus_stp -t <Tcl file> ↵
```

Table 13–14 shows the Tcl commands that you can use with SignalTap II.

Table 13–14. SignalTap II Tcl Commands (Part 1 of 2)

Command	Argument	Description
`open_session`	`-name` <*stp_filename*>	Opens the specified SignalTap II file. All captured data is stored in this file.
`run`	`-instance` <*instance_name*> `-signal_set` <*signal_set*> (optional) `-trigger` <*trigger_name*> (optional) `-data_log` <*data_log_name*> (optional) `-timeout` <*seconds*> (optional)	Starts the analyzer. This command must be followed by all the required arguments to properly start the analyzer. You can optionally specify the name of the data log you want to create. If the Trigger condition is not met, you can specify a timeout value to stop the analyzer.

Table 13–14. SignalTap II Tcl Commands (Part 2 of 2)

Command	Argument	Description
run_multiple_start	None	Defines the start of a set of run commands. Use this command when multiple instances of data acquisition are started simultaneously. Add this command before the set of run commands that specify data acquisition. You must use this command with the run_multiple_end command. If the run_multiple_end command is not included, the run commands will not execute.
run_multiple_end	None	Defines the end of a set of run commands. Use this command when multiple instances of data acquisition are started simultaneously. Add this command after the set of run_commands.
stop	None	Stops data acquisition.
close_session	None	Closes the currently open SignalTap II file. You cannot run the analyzer after the SignalTap II file is closed.



For more information about SignalTap II Tcl commands, refer to the Quartus II Help.

Example 13–4 is an excerpt from a script that is used to continuously capture data. Once the trigger condition is met, the data is captured and stored in the data log.

Example 13–4.

```
#opens signaltap session
open_session -name stp1.stp
#start acquisition of instance auto_signaltap_0 and
#auto_signaltap_1 at the same time
#calling run_multiple_end will start all instances
#run after run_multiple_start call
run_multiple_start
run -instance auto_signaltap_0 -signal_set signal_set_1 -trigger /
trigger_1 -data_log log_1 -timeout 5
run -instance auto_signaltap_1 -signal_set signal_set_1 -trigger /
trigger_1 -data_log log_1 -timeout 5
run_multiple_end
#close signaltap session
close_session
```

Once the script is completed, you should open the SignalTap II file that you used to capture data to examine the contents of the Data log.

Design Example: Using SignalTap II Logic Analyzers in SOPC Builder Systems

Altera Application Note, *AN 323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems* describes how to use the SignalTap II Logic Analyzer to monitor signals located inside a system module generated by SOPC Builder. The system in this example contains many components, including a Nios processor, a direct memory access (DMA) controller, on-chip memory, and an interface to external SDRAM memory. In this example, the Nios processor executes a simple C program from on-chip memory and waits for a button push. After a button is pushed, the processor initiates a DMA transfer, which you analyze using the SignalTap II Logic Analyzer.



For more information about this example and using the SignalTap II Logic Analyzer with SOPC builder systems refer to *AN 323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems* and *AN 446: Debugging NIOS II Systems with the SignalTap II Logic Analyzer*.

Custom Triggering Flow Application Examples

The custom triggering flow in the SignalTap II Logic Analyzer is most useful for organizing a number of triggering conditions and for precise control over the acquisition buffer. This section provides two application examples for defining a custom triggering flow within the SignalTap II Logic Analyzer. Both examples can be easily copied and pasted directly into the state machine description box by using the state display mode **All states in one window**.



For additional triggering flow design examples, refer to the Quartus II On-Chip Debugging Support Resources page for on-chip debugging.

Design Example 1: Specifying a Custom Trigger Position

Actions to the acquisition buffer can accept an optional post-count argument. This post-count argument enables you to define a custom triggering position for each segment in the acquisition buffer. Example 13–5 shows an example that applies a trigger position to all segments in the acquisition buffer. The example describes a triggering flow for an acquisition buffer split into four segments. If each acquisition segment is 64 samples in depth, the trigger position for each buffer will be at sample #34. The acquisition stops after all four segments are filled once.

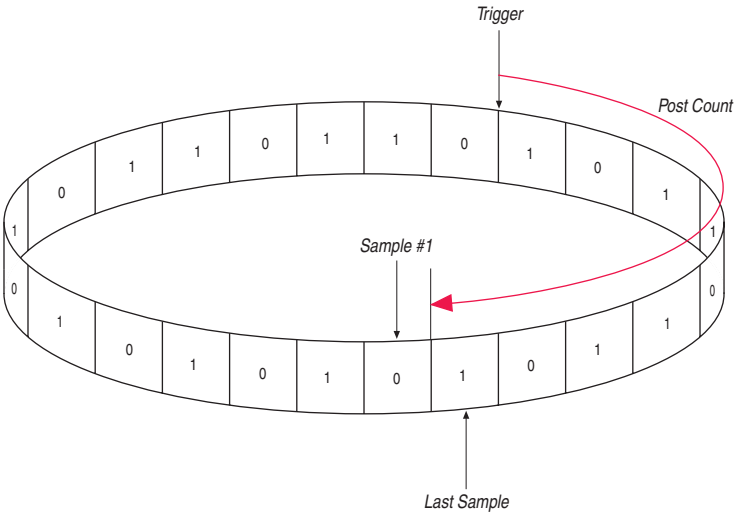
Example 13–5.

```
if (c1 == 3 && condition1)
   trigger 30;
else if ( condition1 )
begin
   segment_trigger 30;
   increment c1;

end
```

Each segment acts as a circular buffer, that continuously updates the memory contents with the signal values. The last acquisition before stopping the buffer is the displayed on the data tab as the last sample number in the affected segment. The trigger position in the affected segment is then defined by *N – post count fill*, where *N* is the number of samples per segment. Figure 13–32 illustrates the triggering position.

Figure 13–32. Specifying a Custom Trigger Position



Design Example 2: Trigger When triggercond1 Occurs Ten Times between triggercond2 and triggercond3

The custom trigger flow description is often useful to count a sequence of events before triggering the acquisition buffer. Example 13–6 on page 13–79 shows such a sample flow. This example uses three basic triggering conditions configured in the SignalTap II setup tab.

This example triggers the acquisition buffer when condition1 occurs after condition3 and occurs ten times prior to condition3. If condition3 occurs prior to ten repetitions of condition1, the state machine transitions to a permanent wait state.

Example 13–6.

```
state ST1:

if ( condition2  )
begin
    reset c1;
    goto ST2;
end


State ST2 :
if ( condition1 )
   increment c1;

else if (condition3 && c1 < 10)
   goto ST3;

else if ( condition3 && c1 >= 10)
   trigger;

ST3:
goto ST3;
```

Conclusion

As the FPGA industry continues to make technological advancements, outdated methodologies need to be replaced with new technologies that maximize productivity. The SignalTap II Logic Analyzer gives you the same benefits as a traditional logic analyzer, without the many shortcomings of a piece of dedicated test equipment. This version of the SignalTap II Logic Analyzer provides many new and innovative features that allow you to capture and analyze internal signals in your FPGA, allowing you to find the source of a design flaw in the shortest amount of time.

Referenced Documents

This chapter references the following documents:

- *AN 323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder System*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 13–15 shows the revision history for this chapter.

Table 13–15. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v.7.2.0	Updated for the Quartus II software version 7.2: • Added new section: "Trigger Condition Flow Control" on page 13–34 • Documented the new feature for State-machine-based triggering • Documented changes to "Using Incremental Compilation with the SignalTap II Logic Analyzer" on page 13–52 • Added additional information about node tappability • Added section "Performance and Resource Considerations" on page 13–55, with information about performance and resource utilization considerations for the SignalTap II Logic Analyzer	Updated for the Quartus II software version 7.2
May 2007 v7.1.0	Added "Referenced Documents" on page 13–71, minor updates to address ADoQS issues.	—
March 2007 v7.0.0	Added Cyclone III device support listed on page 13–4.	—
November 2006 v6.1.0	Updated for the Quartus II software version 6.1: • Updated Figure 13-4, 13-11,13-16, 13-17, 13-18. Added new Figure 13-23. • Miscellaneous changes throughout. • Removed information about incremental routing (feature removed). • Added more detail about the use of incremental compilation. • Added more detail about the use of the Nios II plug-in. • Added more information about SignalTap II file/SOF compatibility. • Updated method for triggering one logic analyzer with another using trigger in/out.	Updated for the Quartus II software version 6.1.
May 2006 v6.0.0	Updated for the Quartus II software version 6.0.	—
October 2005 v5.1.0	Updated for the Quartus II software version 5.1.	—
May 2005 v5.0.0	• Updated information. • Updated figures. • New functionality for Quartus II software 5.0.	—
December 2004 v1.0	Initial release.	—

14. In-System Debugging Using External Logic Analyzers

QII53016-7.2.0

Introduction

The phenomenal growth in design size and complexity continues to make the process of design verification a critical bottleneck for today's FPGA systems. Limited access to internal signals, advanced FPGA packages, and printed circuit board (PCB) electrical noise are all contributing factors in making design debugging and verification the most difficult process of the design cycle. You can easily spend more than 50% of your design cycle time debugging and verifying your design. To help you with the process of design debugging and verification, Altera® provides a solution that allows you to examine the behavior of internal signals using an external logic analyzer and using a minimal number of FPGA I/O pins, while your design is running at full speed on your FPGA.



This chapter's use of 'logic analyzer' includes both logic analyzers and oscilloscopes equipped with digital channels, commonly referred to as mixed signal analyzers or MSOs.

The Logic Analyzer Interface is an application within the Quartus II software used to connect a large set of internal device signals to a small number of output pins. You can connect these output pins to an external logic analyzer for debugging purposes. The Logic Analyzer Interface enables you to connect to and transmit internal signals buried within your FPGA to an external logic analyzer for analysis. The Quartus II Logic Analyzer Interface allows you to debug a large set of internal signals using a small number of output pins. In the Quartus II Logic Analyzer Interface, the internal signals are grouped together, distributed to a user-configurable multiplexer, and then output to available I/O pins on your FPGA. Instead of having a one-to-one relationship between internal signals to output pins, the Quartus II Logic Analyzer Interface enables you map many internal signals to a smaller number of output pins. The exact number of internal signals that you can map to an output pin varies based on the multiplexer settings in the Quartus II Logic Analyzer Interface.

Optionally, you can use the Logic Analyzer Interface with the Quartus II Incremental Compilation.

Choosing a Logic Analyzer

During the debugging phase of your project, you have the choice of using:

- SignalTap® II, the embedded logic analyzer.
- An external logic analyzer, which connects to internal signals in your FPGA, by using the Quartus II Logic Analyzer Interface.

Table 14–1 describes the advantages to both debugging technologies.

Table 14–1. Comparing the SignalTap II Embedded Logic Analyzer with the Logic Analyzer Interface

Feature	Logic Analyzer Interface	SignalTap II Embedded Logic Analyzer
Sample Depth—You will have access to a wider sample depth with an external logic analyzer. In SignalTap II, the maximum sample depth is set to 128 Kb, which is a device constraint. However, with an external logic analyzer, there are no device constraints, providing you a wider sample depth.	✓	—
Debugging Timing Issues—Using an external logic analyzer provides you with access to a "timing" mode, which enables you to debug combined streams of data.	✓	—
Performance—You frequently have limited routing resources available to place-and-route when you use SignalTap II with your design. An external logic analyzer adds minimal logic, which removes resource limits on place-and-route.	✓	—
Triggering Capability—Although advanced triggering is available in SignalTap II, many additional triggering options are available on an external logic analyzer.	✓	—
Use of Output Pins—Using the SignalTap II Logic Analyzer, no additional output pins are required. Using an external logic analyzer requires the use of additional output pins.	—	✓
Acquisition Speed—With the SignalTap II Logic Analyzer, you can acquire data at speeds of over 200 MHz. You can achieve the same acquisition speeds with an external logic analyzer, however you have to consider signal integrity issues.	—	✓

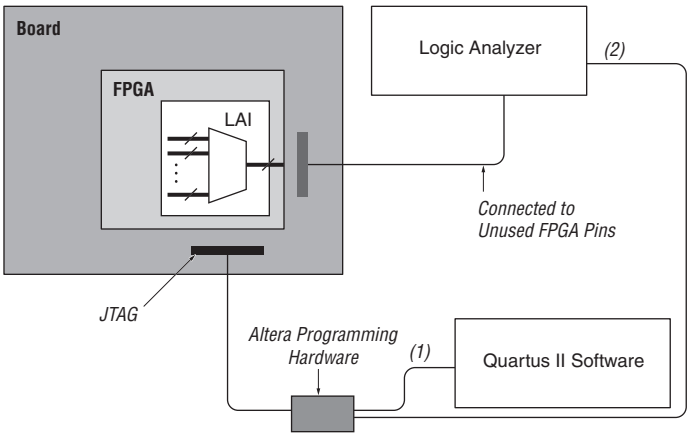
Required Components

You must have the following components to perform analysis using the Quartus II Logic Analyzer Interface:

- The Quartus II software starting with version 5.1 and later
- The device under test
- An external logic analyzer
- An Altera communications cable
- A cable to connect the FPGA to the external logic analyzer

Figure 14–1 shows the Logic Analyzer Interface and the hardware setup.

Figure 14–1. Logic Analyzer Interface and Hardware Setup



Notes to Figure 14–1:

(1) Configuration and control of the LAI using computer loaded with Quartus II via the JTAG port.

(2) Configuration and control of the LAI using a third-party vendor logic analyzer via the JTAG port. Support varies by vendor.

FPGA Device Support

You can use the Quartus II Logic Analyzer Interface with the following FPGA device families:

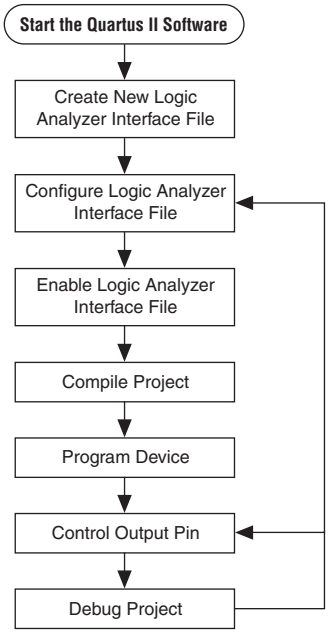
- Arria™ GX
- Stratix® III
- Stratix II
- Stratix II GX
- Stratix
- Stratix GX

- Cyclone® III
- Cyclone II
- Cyclone
- MAX® II
- APEX™ 20K
- APEX II

Debugging Your Design Using the Logic Analyzer Interface

Figure 14–2 shows the steps you must follow to debug your design with the Quartus II Logic Analyzer Interface.

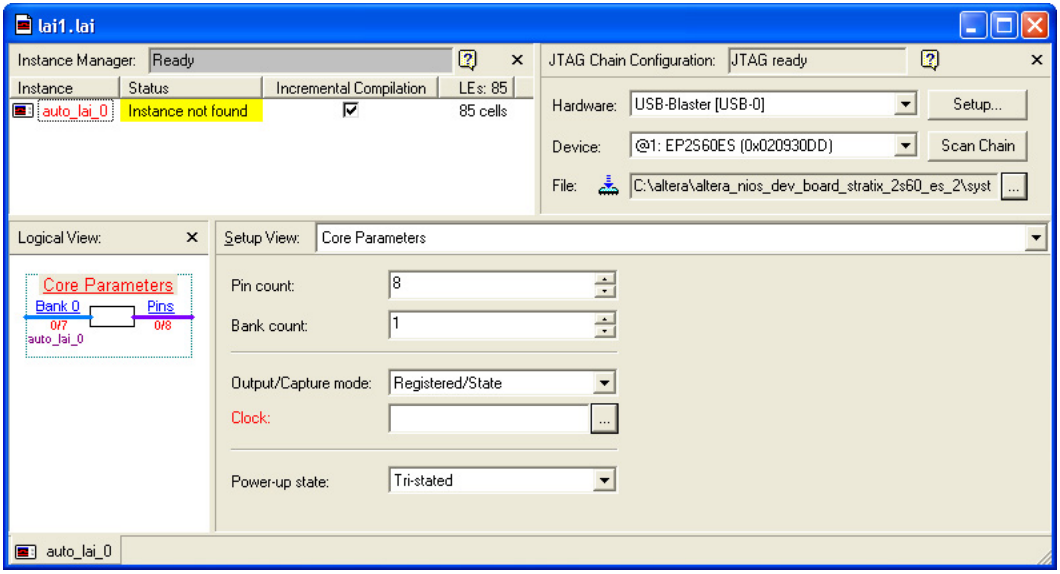
Figure 14–2. Logic Analyzer Interface Process Flow



Creating a Logic Analyzer Interface File

The Logic Analyzer Interface File (**.lai**), defines the interface that builds a connection between internal FPGA signals and your external logic analyzer. An example of a Logic Analyzer Interface File is shown in Figure 14–3.

Figure 14–3. Example of a Logic Analyzer Interface Editor



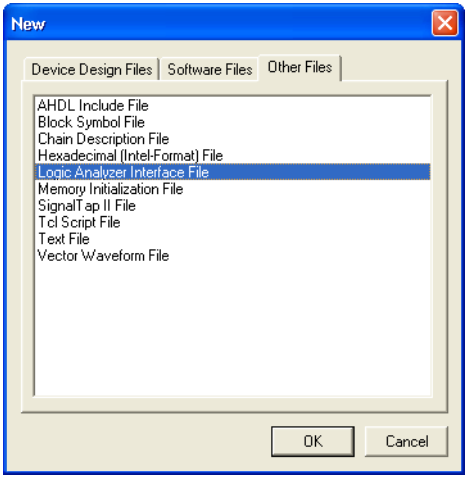
To define the Quartus II Logic Analyzer Interface, you can create a new Logic Analyzer Interface File or use an existing Logic Analyzer Interface File.

Creating a New Logic Analyzer Interface File

To create a new Logic Analyzer Interface File, perform the following steps:

1. In the Quartus II software, on the File menu, click **New**. The **New** dialog box opens.
2. Click the **Other Files** tab and select **Logic Analyzer Interface File** (Figure 14–4).

Figure 14–4. Creating a New Logic Analyzer File



3. Click **OK**. The Logic Analyzer Interface editor opens. The file name is assigned by the Quartus II software (refer to Figure 14–3 on page 14–5). When you save the file, you will be prompted for a file name. Refer to "Saving the External Analyzer Interface File" on page 14–7.

Opening an Existing External Analyzer Interface File

To open an existing Logic Analyzer Interface File, on the Tools menu, click **Logic Analyzer Interface Editor**. If no Logic Analyzer Interface File is enabled for the current project, the editor automatically creates a new Logic Analyzer Interface File. If a Logic Analyzer Interface File is currently enabled for the project, that file opens when you select the **Logic Analyzer Interface Editor**.

Another way to open an existing Logic Analyzer Interface File is on the File Menu, click **Open**, and select the Logic Analyzer Interface File you want to open.

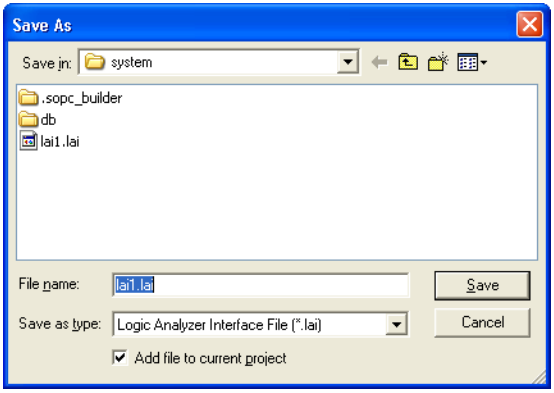
Saving the External Analyzer Interface File

To save your Logic Analyzer Interface File, perform the following steps:

1. In the Quartus II software, on the File menu, click **Save As**, The **Save As** dialog box opens (Figure 14–5).

2. In the **File name** box, enter the desired file name. Click Save (Figure 14–5).

Figure 14–5. Saving the Logic Analyzer Interface File



Configuring the Logic Analyzer Interface File Core Parameters

After you have created your Logic Analyzer Interface File, you must configure the Logic Analyzer Interface File core parameters.

To configure the Logic Analyzer Interface File core parameters, select Core Parameters from the Setup View list. Refer to Figure 14–6.

Figure 14–6. Logic Analyzer Interface File Core Parameters

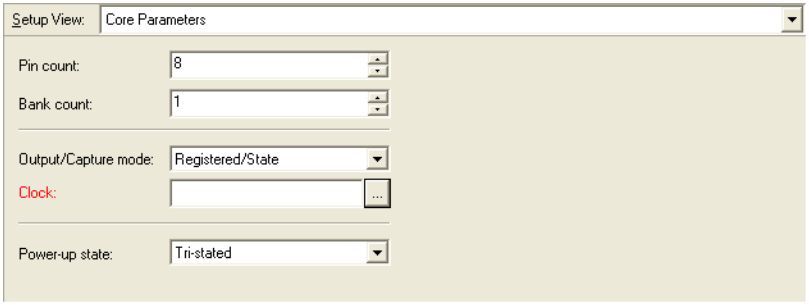


Table 14–2 lists the Logic Analyzer Interface File core parameters.

Table 14–2. Logic Analyzer Interface File Core Parameters (Part 1 of 2)

Parameter	Description
Pin Count	The Pin Count parameter signifies the number of pins you want dedicated to your Logic Analyzer Interface. The pins need to be connected to a debug header on your board. Within the FPGA, each pin is mapped to a user-configurable number of internal signals. The Pin Count parameter can range from 1 to 256 pins.
Bank Count	The Bank Count parameter signifies the number of internal signals that you want to map to each pin. For example, a Bank Count of 8 implies that you will connect eight internal signals to each pin. The Bank Count parameter can range from 1 to 256 banks.
Output/ Capture Mode	The Output/Capture Mode parameter signifies the type of acquisition you perform. There are two options that you can select: Combinational/Timing—This acquisition uses your external logic analyzer's internal clock to determine when to sample data. Because Combinational/Timing acquisition samples data asynchronously to your FPGA, you need to properly determine the sample frequency you should use to debug and verify your system. This mode is effective if you want to measure timing information such as channel-to-channel skew. For more information on the sampling frequency, and what speeds it can run at refer to the data sheet for your external logic analyzer. Registered/State—This acquisition uses a signal from your system under test to determine when to sample. Because Registered/State acquisition samples data synchronously with your FPGA, it provides you with a functional view of your FPGA while it is running. This mode is effective when you want to verify the functionality of your design.

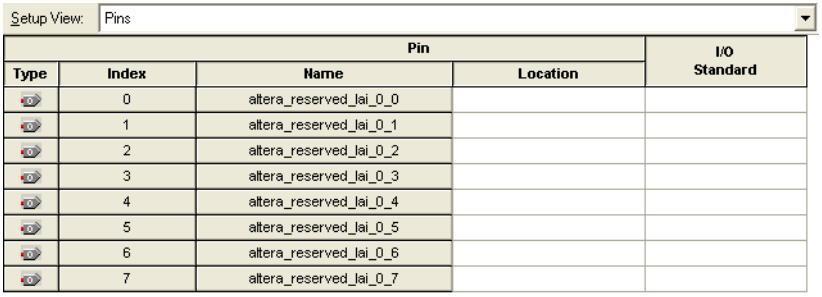
Table 14–2. Logic Analyzer Interface File Core Parameters (Part 2 of 2)

Parameter	Description
Clock	The clock parameter is available only when Output/Capture Mode is set to Registered State. You must specify the sample clock in the Core Parameters view. The sample clock can be any signal in your design. However, for best results, Altera recommends that you use a clock with an operating frequency fast enough to sample the data you would like to acquire.
Power-Up State	The Power-Up State parameter specifies the power-up state of the pins you have designated for use with the Logic Analyzer Interface. You have the option of selecting tri-stated for all pins, or selecting a particular bank that you have enabled.

Mapping the Logic Analyzer Interface File Pins to Available I/O Pins

To configure the Logic Analyzer Interface File I/O pins parameters, select Pins from the Setup View list (Figure 14–7).

Figure 14–7. Logic Analyzer Interface File Pins Parameters



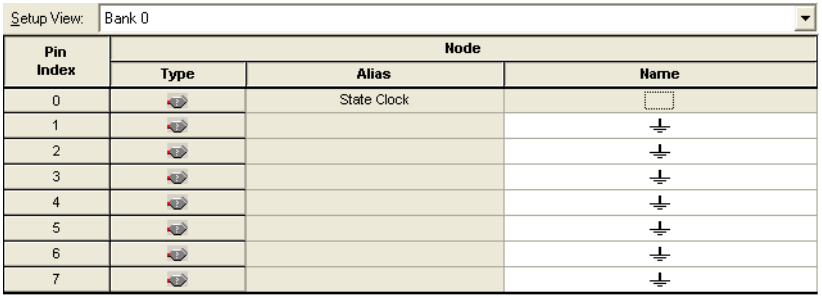
To assign pin locations for the Logic Analyzer Interface, double-click the **Location** column next to the reserved pins in the **Names** column. This opens the Pin Planner.

For information on how to use the Pin Planner, refer to the *Pin Planner* section in the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Mapping Internal Signals to the Logic Analyzer Interface Banks

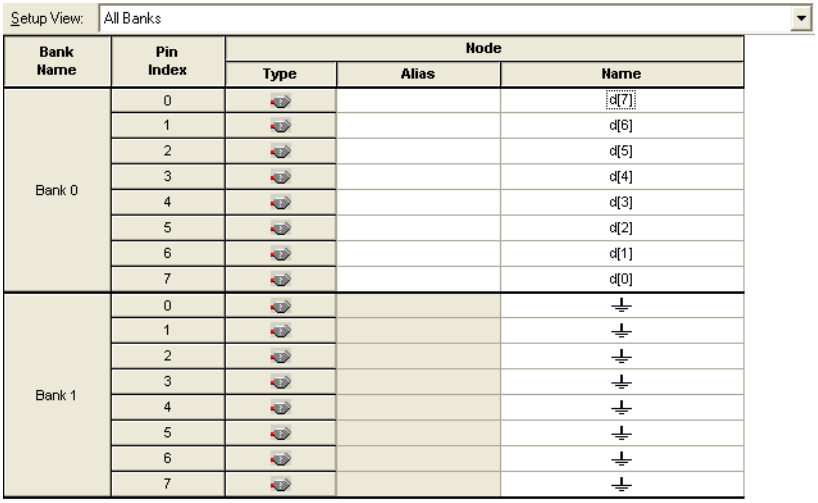
After you have specified the number of banks to use in the Core Parameters settings page, you must assign internal signals for each bank in the Logic Analyzer Interface. Click the **Setup View** arrow and select **Bank *n*** or **ALL Banks** (Figure 14–8).

Figure 14–8. Logic Analyzer Interface Bank Parameters



To view all of your bank connections, click **Setup View** and select **All Banks** (Figure 14–9).

Figure 14–9. Logic Analyzer Interface All Bank Parameters



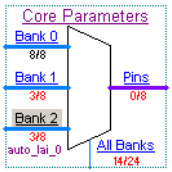
Using the Node Finder

Before making bank assignments, on the View menu, point to Utility Windows, and click **Node Finder**. Find the signals that you want to acquire, then drag and drop the signals from the Node Finder dialog box

into the bank **Setup View**. When adding signals, use **SignalTap II: pre-synthesis** for non-incrementally routed instances and **SignalTap II: post-fitting** for incrementally routed instances.

As you continue to make assignments in the bank Setup View, the schematic of your Logic Analyzer Interface in the Logical View of your Logic Analyzer Interface File begins to reflect your assignments (Figure 14–10).

Figure 14–10. A Logical View of the Logic Analyzer Interface Schematic



Continue making assignments for each bank in the **Setup View** until you have added all of the internal signals for which you wish to acquire data.

You can right-click to switch between the Logic Analyzer Interface schematic and the Logic Analyzer Interface Setup view.

Enabling the Logic Analyzer Interface Before Compiling Your Quartus II Project

Compile your project after you have completed the following steps:

- Configure your Logic Analyzer Interface parameters
- Map the Logic Analyzer Interface pins to available I/O pins
- Map the internal signals to the Logic Analyzer Interface banks

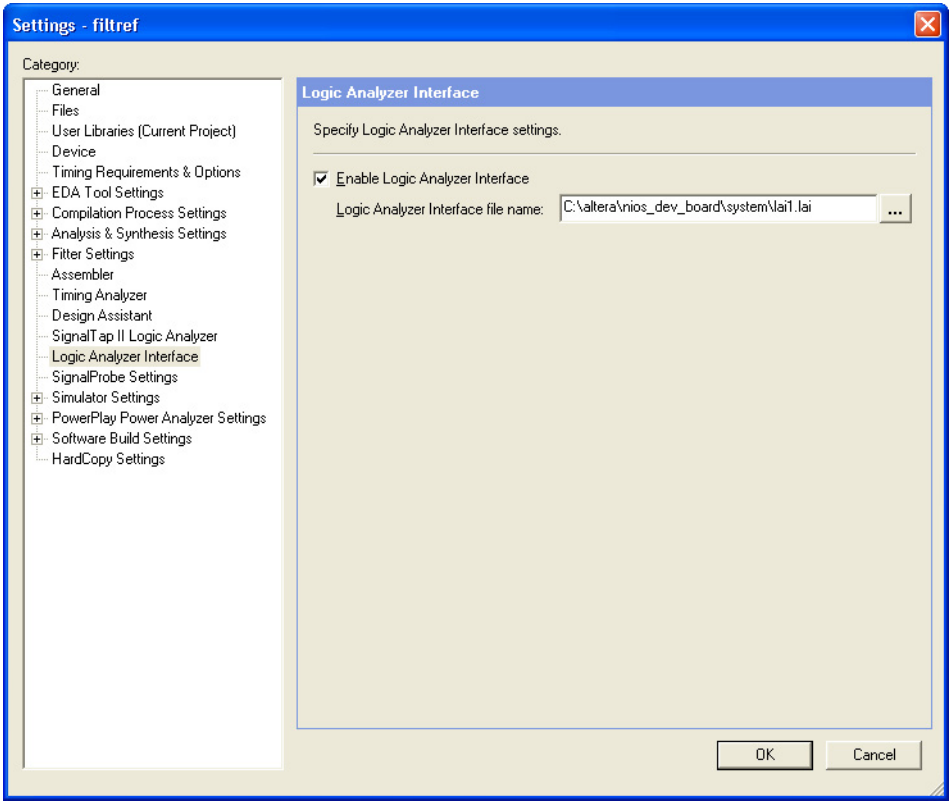
Compiling Your Quartus II Project

Before compilation, you must enable the Logic Analyzer Interface.

1. On the Assignments menu, click **Settings**. The **Settings** dialog box opens. Under Category, click **Logic Analyzer Interface**. The **Logic Analyzer Interface** displays. Turn on **Enable Logic Analyzer Interface.**

2. Click **Logic Analyzer Interface file name** and specify the full path name to your Logic Analyzer Interface File (Figure 14–11).

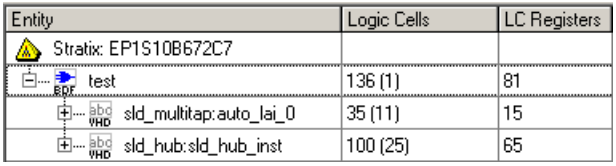
Figure 14–11. Settings Dialog Box—Logic Analyzer Interface Settings



After you have specified the name of your Logic Analyzer Interface File, you must compile your project. To compile your project, on the Processing menu, click **Start Compilation**.

To ensure the Logic Analyzer Interface is properly compiled with your project, expand the entity hierarchy in the Project Navigator. (To display the Project Navigator, on the View menu, point to **Utility Windows**, and click **Project Navigator**.) If the Logic Analyzer Interface compiled with your design, the `sld_hub` and `sld_multitap` entities are shown in the project navigator.

Figure 14–12. Project Navigator

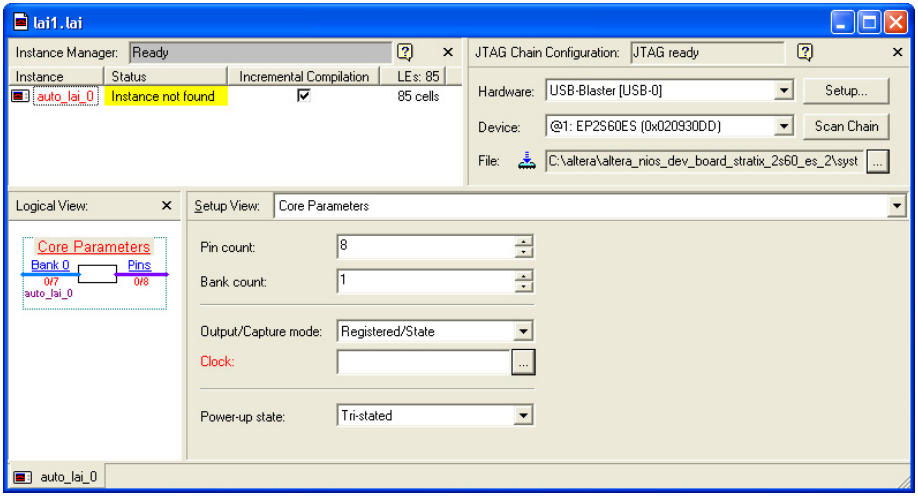


Programming Your FPGA Using the Logic Analyzer Interface

After compilation completes, you must configure your FPGA before using the Logic Analyzer Interface. To configure a device for use with the Logic Analyzer Interface, follow these steps:

1. Open the Logic Analyzer Interface File Editor (Figure 14–13).
2. Under **JTAG Chain Configuration**, click **Hardware** and select your hardware communications device. You may have to click **Settings** to configure your hardware.
3. Click **Device** and select the FPGA device to which you want to download the design (it may be automatically detected). You may have to click **Scan Chain** to configure your device.
4. Click **File** and select the SRAM Object File (**.sof**) that includes the Logic Analyzer Interface File (it may be automatically detected).
5. If desired, turn on **Incremental Compilation**.
6. Save the Logic Analyzer Interface File.
7. Click the **Program Device** icon to program the device.

Figure 14–13. The JTAG Section of the Logic Analyzer Interface File



Using the Logic Analyzer Interface with Multiple Devices

You can use the Logic Analyzer Interface with multiple devices in your JTAG chain. Your JTAG chain can also consist of devices that do not support the Logic Analyzer Interface or non-Altera, JTAG-compliant devices. To use the Logic Analyzer Interface in more than one FPGA, create a Logic Analyzer Interface and configure a Logic Analyzer Interface File for each FPGA that you want to analyze. To perform multi-FPGA analysis, perform the following steps:

1. Open the Quartus II software.
2. Create, configure, and compile a Logic Analyzer Interface File for each design.
3. Open one Logic Analyzer Interface File at a time.

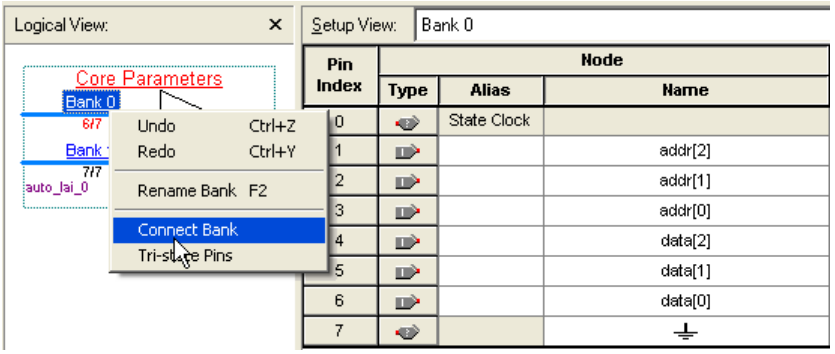
   You do not have to open a Quartus II project to open a Logic Analyzer Interface File.

4. Follow Steps 2 through 6 under "Programming Your FPGA Using the Logic Analyzer Interface" on page 14–13.
5. Click the **Program Device** icon to program the device.
6. Control each Logic Analyzer Interface File independently.

Configuring Banks in the Logic Analyzer Interface File

When you have programmed your FPGA, you can control which bank is mapped to the reserved Logic Analyzer Interface File output pins. To control which bank is mapped, right-click on the bank in the schematic in the logical view and click **Connect Bank**.

Figure 14–14. Configuring Banks



Acquiring Data on Your Logic Analyzer

To acquire data on your logic analyzer, you must establish a connection between your device and the external logic analyzer.

For more information on this process, and for guidelines on how to establish connections between debugging headers and logic analyzers, refer to the documentation for your logic analyzer.

Advanced Features

This section describes the following advanced features:

- Using the Logic Analyzer Interface with Incremental Compilation
- Creating Multiple Logic Analyzer Interface Instances in One FPGA

Using the Logic Analyzer Interface with Incremental Compilation

Using the Logic Analyzer Interface with Incremental Compilation enables you to preserve the synthesis and fitting of your original design and add the Logic Analyzer Interface to your design without recompiling your original source code.

To use the Logic Analyzer Interface with Incremental Compilation, perform the following steps:

1. Start the Quartus II software.

2. Enable Design Partitions. To enable Partitions, perform the following steps:

    a. On the Assignments menu, click, **Design Partitions**.

    b. In the **Incremental Compilation** list, select **Full Incremental Compilation**.

    c. Create Design Partitions for the entities in your design, and set the Netlist Type to **Post-fit**.

    d. On the Processing menu, click **Start Compilation**.

3. Enable Logic Analyzer Interface Incremental Compilation by performing these steps:

    a. In your Logic Analyzer Interface File, under **Instance Manager**, click **Incremental Compilation**.

    When you enable Incremental Compilation, all existing presynthesis signals will be converted into post-fitting signals. Only post-fitting signals can be used with the Logic Analyzer Interface with Incremental Compilation.

    b. Add Post-Fitting nodes to your Logic Analyzer Interface File.

    c. On the Processing menu, click **Start Compilation**.

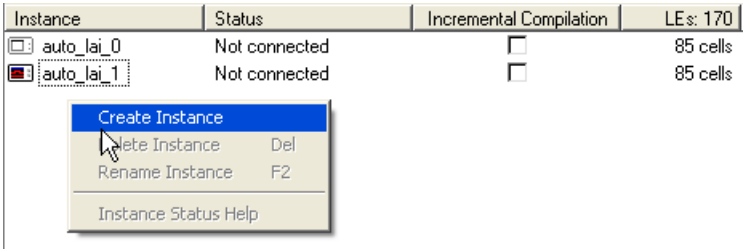
Creating Multiple Logic Analyzer Interface Instances in One FPGA

The Logic Analyzer Interface includes support for multiple interfaces in one FPGA. This feature is particularly useful when you want to build Logic Analyzer Interface configurations that contain different settings. For example, you can build one Logic Analyzer Interface instance to perform Registered/State analysis and build another instance that performs Combinational/Timing analysis on the same set of signals.

Another example would be if you want to perform Registered/State analysis on portions of your design that are in different clock domains.

To create multiple Logic Analyzer Interfaces, on the Edit menu, click **Create Instance**. Alternatively, you can right-click in the Instance Manager window, and click **Create Instance**.

Figure 14–15. Creating Multiple Logic Analyzer Interface Instances in One FPGA



Conclusion

As the FPGA industry continues to make technological advancements, outdated debugging methodologies must be replaced with new technologies that maximize productivity. The Logic Analyzer Interface feature enables you to connect many internal signals within your FPGA to an external logic analyzer with the use of a small number of I/O pins. This new technology in the Quartus II software enables you to use feature-rich external logic analyzers to debug your FPGA design, ultimately enabling you to delver your product in the shortest amount of time.

Document Revision History

Table 14–3 shows the revision history for this chapter.

Table 14–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No changes to content.	—
May 2007 v7.1.0	Minor updates to address ADoQS issues.	—
March 2007 v7.0.0	Added Cyclone III device support listed on page 14–3.	—
November 2006 v6.1.0	Added new revision history table format to this document.	—
May 2006 v6.0.0	Chapter title changed. Minor updates for the Quartus II software version 6.0.0.	—
October 2005 v5.1.0	Initial release.	—

15. In-System Updating of Memory and Constants

QII53012-7.2.0

Introduction

FPGA designs are growing larger in density and are becoming more complex. Designers and verification engineers require more access to the design that is programmed in the device to quickly identify, test, and resolve issues. The in-system updating of memory and constants capability of the Quartus® II software provides read and write access to in-system FPGA memories and constants through the Joint Test Action Group (JTAG) interface, making it easier to test changes to memory contents in the FPGA while the FPGA is functioning in the end system.

This chapter explains how to use the Quartus II In-System Memory Content Editor as part of your FPGA design and verification flow.

This chapter contains the following sections:

- "Device Megafunction Support" on page 15–2
- "Using In-System Updating of Memory Constants with Your Design" on page 15–3
- "Creating In-System Modifiable Memories Constants" on page 15–3
- "Running the In-System Memory Content Editor" on page 15–4

Overview

The ability to read and update memories and constants in a programmed device provides more insight into and control over your design. The Quartus II In-System Memory Content Editor gives you access to device memories and constants. When used in conjunction with the SignalTap® II embedded logic analyzer, this feature provides you the visibility required to debug your design in the hardware lab.



For more information on the SignalTap II embedded logic analyzer, refer to the *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

The ability to read data from memories and constants allows you to quickly identify the source of problems. In addition, the write capabilities allow you to bypass functional issues by writing expected data. For example, if a parity bit in your memory is incorrect, you can use the In-System Content Editor to write the correct parity bit values into your RAM, allowing your system to continue functioning. You can also intentionally write incorrect parity bit values into your RAM to check your design's error handling functionality.

Device Megafunction Support

The following tables list the devices and types of memories and constants that are currently supported by the Quartus II software. Table 15–1 lists the types of memory supported by the MegaWizard® Plug-In Manager and the In-System Memory Content Editor.

Table 15–1. MegaWizard Plug-In Manager Support

Installed Plug-Ins Category	Megafunction Name
Gates	LPM_CONSTANT
Memory Compiler	RAM: 1-PORT, ROM: 1-PORT
Storage	ALTSYNCRAM, LPM_RAM_DQ, LPM_ROM

Table 15–2 lists support for in-system updating of memory and constants for the Stratix® series, Arria™ GX, Cyclone® series, APEX™ II, APEX 20K, and Mercury™ device families.

Table 15–2. Supported Megafunctions

MegaFunction	Arria GX / Stratix Series			Cyclone Series	APEX II	APEX 20K	Mercury
	M512 Blocks	M4K Blocks	MegaRAM Blocks				
LPM_CONSTANT	Read/ Write	Read/ Write	Read/ Write	Read/ Write	Read/ Write	Read/ Write	Read/ Write
LPM_ROM	Write	Read/ Write	N/A	Read/ Write	Read/ Write	Write	Read/ Write
LPM_RAM_DQ	N/A	Read/ Write	Read/ Write	Read/ Write	Read/ Write	N/A *(1)*	Read/ Write
ALTSYNCRAM (ROM)	Write	Read/ Write	N/A	Read/ Write	N/A	N/A	N/A
ALTSYNCRAM (Single-Port RAM Mode)	N/A	Read/ Write	Read/ Write	Read/ Write	N/A	N/A	N/A

Note to Table 15–2:

(1) Only write-only mode is applicable for this single-port RAM. In read-only mode, use LPM_ROM instead of LPM_RAM_DQ.

Using In-System Updating of Memory Constants with Your Design

Using the In-System Updating of Memory and Constants feature requires the following steps:

1. Identify the memories and constants that you want to access.
2. Edit the memories and constants to be run-time modifiable.
3. Perform a full compilation.
4. Program your device.
5. Launch the In-System Memory Content Editor.

Creating In-System Modifiable Memories Constants

When you specify that a memory or constant is run-time modifiable, the Quartus II software changes the default implementation. A single-port RAM is converted to dual-port RAM, and a constant is implemented in registers instead of look-up tables (LUTs). These changes enable run-time modification without changing the functionality of your design. For a list of run-time modifiable megafunctions, refer to Table 15–1.

To enable your memory or constant to be modifiable, perform the following steps:

1. On the Tools menu, click **MegaWizard Plug-In Manager**.
2. If you are creating a new megafunction, select **Create a new custom megafunction variation**. If you have an existing megafunction, select **Edit an existing custom megafunction variation**.
3. Make the necessary changes to the megafunction based on the characteristics required by your design, turn on **Allow In-System Memory Content Editor to capture and update content independently of the system clock** and type a value in the **Instance ID** text box. These parameters can be found on the last page of the wizard for megafunctions that support in-system updating.

   The Instance ID is a four-character string used to distinguish the megafunction from other in-system memories and constants.

4. Click **Finish**.
5. On the Processing menu, click **Start Compilation**.

If you instantiate a memory or constant megafunction directly using ports and parameters in VHDL or Verilog HDL, add or modify the `lpm_hint` parameter as shown below.

In VHDL code, add the following:

```
lpm_hint => "ENABLE_RUNTIME_MOD = YES,
  INSTANCE_NAME = <instantiation name>";
```

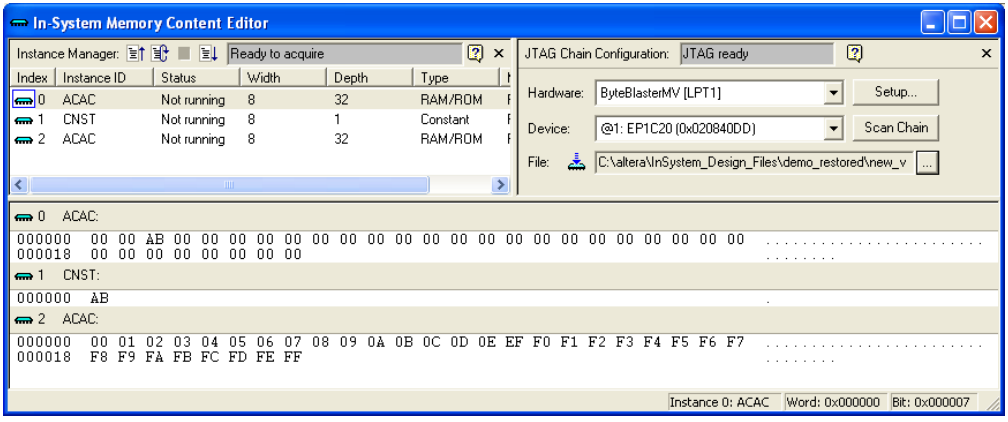
In Verilog HDL code, add the following:

```
defparam <megafunction instance name>.lpm_hint =
  "ENABLE_RUNTIME_MOD = YES,
  INSTANCE_NAME = <instantiation name>";
```

Running the In-System Memory Content Editor

The In-System Memory Content Editor is separated into the Instance Manager, JTAG Chain Configuration, and the Hex Editor (Figure 15–1).

Figure 15–1. In-System Memory Content Editor



The Instance Manager displays all available run-time modifiable memories and constants in your FPGA device. The JTAG Chain Configuration section allows you to program your FPGA and select the Altera® device in the chain to update.

Using the In-System Memory Content Editor does not require that you open a project. The In-System Memory Content Editor retrieves all instances of run-time configurable memories and constants by scanning the JTAG chain and sending a query to the specific device selected in the JTAG Chain Configuration section.

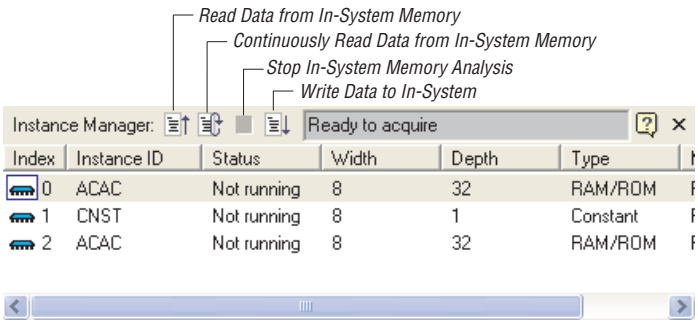
Each In-System Memory Content Editor can access the in-system memories and constants in a single device. If you have more than one device containing in-system configurable memories or constants in a JTAG chain, you can launch multiple In-System Memory Content Editors within the Quartus II software to access the memories and constants in each of the devices.

Instance Manager

Scan the JTAG chain to update the Instance Manager with a list of all run-time modifiable memories and constants in the design. The Instance Manager displays the Index, Instance, Status, Width, Depth, Type, and Mode of each element in the list.

You can read and write to in-system memory using the Instance Manager as shown in Figure 15–2.

Figure 15–2. Instance Manager Controls



The following buttons are provided in the Instance Manager:

- **Read data from In-System Memory**—reads the data from the device independently of the system clock and displays it in the Hex Editor
- **Continuously Read Data from In-System Memory**—Continuously reads the data asynchronously from the device and displays it in the Hex Editor
- **Stop In-System Memory Analysis**—Stops the current read or write operation

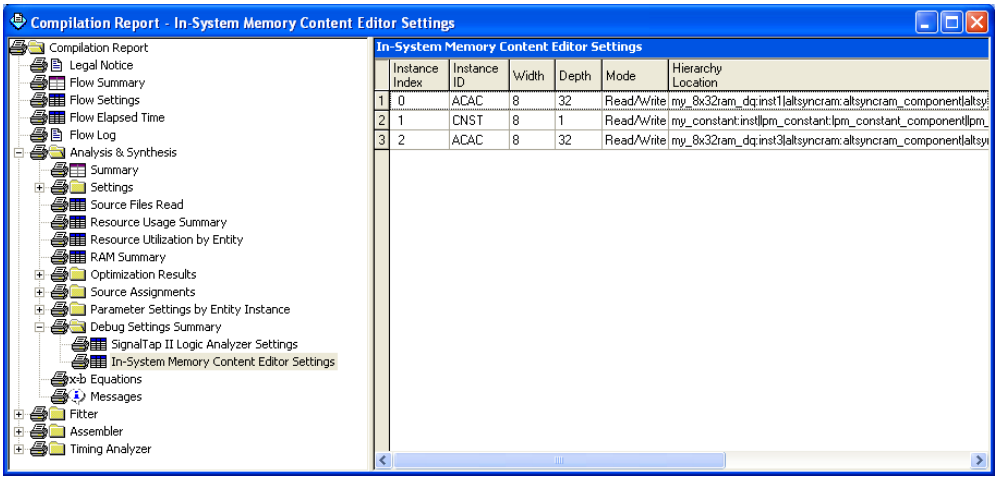
- **Write Data to In-System Memory**—Asynchronously writes data present in the Hex Editor to the device

In addition to the buttons available in the Instance Manager, you can also read and write data by selecting the command from the Processing menu, or the right button pop-up menu in the Instance Manager or Hex Editor.

The status of each instance is also displayed beside each entry in the Instance Manager. The status indicates if the instance is **Not running**, **Offloading data** or **Updating Data**. The health monitor provides useful information about the status of the editor.

The Quartus II software assigns a different index number to each in-system memory and constant to distinguish between multiple instances of the same memory or constant function. View the **In-System Memory Content Editor Setting** section of the compilation report to match an index with the corresponding instance ID (Figure 15–3).

Figure 15–3. Compilation Report In-System Memory Content Editor Setting Section



Editing Data Displayed in the Hex Editor

You can edit the data read from your in-system memories and constants displayed in the Hex Editor by typing values directly into the editor or by importing memory files.

To modify the data displayed in the Hex Editor, click a location in the editor and type or paste in the new data. The new data appears as blue indicating modified data that has not been written into the FPGA. On the Edit menu, choose Value, and click **Fill with 0's**, **Fill with 1's**, **Fill with Random Values**, or **Custom Fills to** update a block of data by selecting a block of data.

Importing Exporting Memory Files

Importing and exporting memory files lets you quickly update in-system memories with new memory images and record data for future use and analysis.

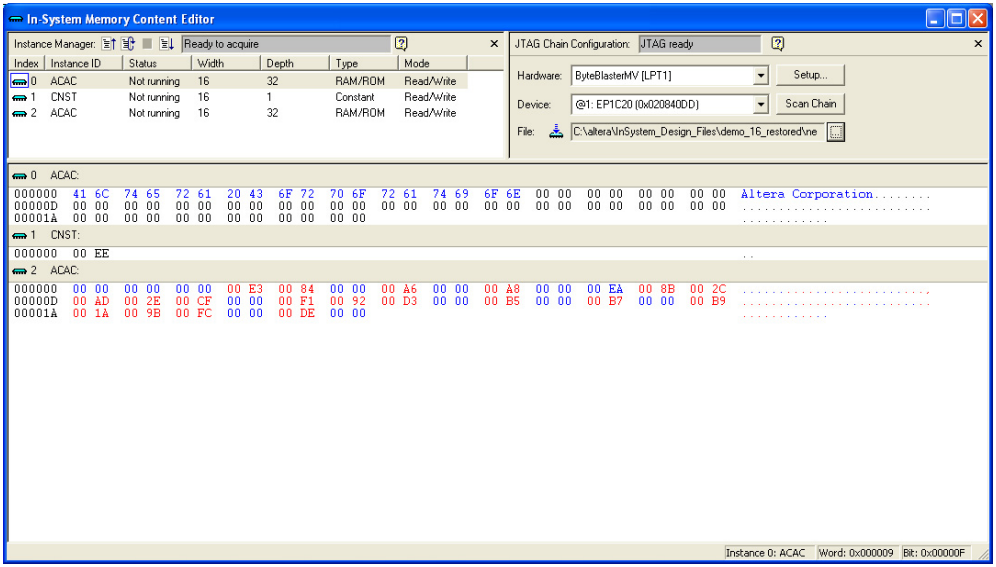
On the Edit menu, click **Import Data from File** to import a memory file, select an in-system memory or constant from the instance manager. You can only import a memory file that is in either a Hexadecimal (Intel-Format) file (**.hex**) or memory initialization file (**.mif**) format.

On the Edit menu, click **Export Data to File** to export data displayed in the Hex Editor to a memory file, to select an in-system memory or constant from the instance manager. You can export data to a **.hex**, **.mif**, Verilog Value Change Dump file (**.vcd**), or RAM Initialization file (**.rif**) format.

Viewing Memories Constants in the Hex Editor

For each instance of an in-system memory or constant, the Hex Editor displays data in hexadecimal representation and ASCII characters (if the word size is a multiple of 8 bits). The arrangement of the hexadecimal numbers depends on the dimensions of the memory. For example, if the word width is 16 bits, the Hex Editor displays data in columns of words that contain columns of bytes (Figure 15–4).

Figure 15–4. Editing 16-Bit Memory Words Using the Hex Editor



Unprintable ASCII characters are represented by a period (.). The color of the data changes as you perform reads and writes. Data displayed in black indicates the data in the Hex Editor was the same as the data read from the device. If the data in the Hex Editor changes color to red, the data previously shown in the Hex Editor was different from the data read from the device.

As you analyze the data, you can use the cursor and the status bar to quickly identify the exact location in memory. The status bar is located at the bottom of the In-System Memory Content Editor and displays the selected instance name, word position, and bit offset (Figure 15–5).

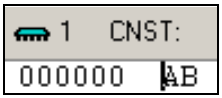
Figure 15–5. Status Bar in the In-System Memory Content Editor



The bit offset is the bit position of the cursor within the word. In the following example, a word is set to be 8-bits wide.

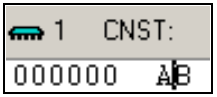
With the cursor in the position shown in Figure 15–6, the word location is 0x0000 and the bit position is 0x0007.

Figure 15–6. Hex Editor Cursor Positioned at Bit 0×0007



With the cursor in the position shown in Figure 15–7, the word location remains 0x0000 but the bit position is 0x0003.

Figure 15–7. Hex Editor Cursor Positioned at Bit 0×0003



Scripting Support

The In-System Memory Content Editor supports reading and writing of memory contents via a Tcl script or Tcl commands entered in a command prompt. For detailed information about scripting command options, refer to the Quartus II command-line and Tcl API Help browser.

To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp r
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF form.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

The commonly used commands for the In-System Memory Content Editor are as follows:

- Reading from memory:
  `read_content_from_memory`
  `[-content_in_hex]`
  `-instance_index` <*instance index*>
  `-start_address` <*starting address*>
  `-word_count` <*word count*>

- Writing to memory:
  `write_content_to_memory`
- Save memory contents to file:
  `save_content_from_memory_to_file`
- Update memory contents from File:
  `update_content_to_memory_from_file`



For descriptions about the command options and scripting examples, refer to the Tcl API Help Browser and the *Quartus II Scripting Reference Manual*.

Programming the Device Using the In-System Memory Content Editor

If you make changes to your design, you can program the device from within the In-System Memory Content Editor. To program the device, follow these steps:

1. On the Tools menu, click **In-System Memory Content Editor**.
2. In the **JTAG Chain Configuration** panel of the In-System Memory Content Editor, select the SRAM object file (**.sof**) that includes the modifiable memories and constants.
3. Click **Scan Chain**.
4. In the **Device** list, select the device you want to program.
5. Click **Program Device**.

Example: Using the In-System Memory Content Editor with the SignalTap II Embedded Logic Analyzer

The following scenario describes how you can use the In-System Updating of Memory and Constants feature with the SignalTap II embedded logic analyzer to efficiently debug your design in-system. Although both the In-System Content Editor and the SignalTap II embedded logic analyzer use the JTAG communication interface, you are able to use them simultaneously.

After completing your FPGA design, you find that the characteristics of your FIR filter design are not as expected.

1. To locate the source of the problem, change all your FIR filter coefficients to be in-system modifiable and instantiate the SignalTap II embedded logic analyzer.

2. Using the SignalTap II embedded logic analyzer to tap and trigger on internal design nodes, you find the FIR filter to be functioning outside of the expected cut-off frequency.

3. Using the **In-System Memory Content Editor**, you check the correctness of the FIR filter coefficients. Upon reading each coefficient, you discover that one of the coefficients is incorrect.

4. Since your coefficients are in-system modifiable, you update the coefficients with the correct data using the **In-System Memory Content Editor**.

In this scenario, you are able to quickly locate the source of the problem using both the In-System Memory Content Editor and the SignalTap II embedded logic analyzer. You are also able to verify the functionality of your device by changing the coefficient values before modifying the design source files.

An extension to this example would be to modify the coefficients with the In-System Memory Content Editor to vary the characteristics of the FIR filter (for example, filter attenuation, transition bandwidth, cut-off frequency, and windowing function).

Conclusion

The In-System Updating of Memory and Constants feature provides access into a device for efficient debug in a hardware lab. You can use In-System Memory Updating of Memory and Constants with the SignalTap II embedded logic analyzer to maximize the visibility into an Altera FPGA. By increasing visibility and access to internal logic of the device, you are able to more quickly identify and resolve problems with your design or its implementation.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 15–3 shows the revision history of this chapter.

Table 15–3. Document Revision History

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 15–11.	—
May 2007 v7.1.0	• Added Scripting Support section on page 15–9. • Added Referenced Documents on page 15–11.	Updates made for Quartus II version 7.1.
March 2007 v7.0.0	Added Cyclone III device support listed on page 15–2.	—
November 2006 v6.1.0	• Added revision history to the document. • Updated Table 15–2.	Added information for Stratix III support.
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0	—
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 13 was formerly Chapter 12 in version 5.0.	—
May 2005 v5.0.0	• Chapter 12 was formerly in Section V of Vol 3 in 4.2.	—
December 2004 v1.2	• Chapter 12 was formerly Chapter 11. • Updated tables. • Corrected the Verilog code for the `lpm_hint` parameter. • Re-organized the "Making Changes" segment into the Editing Data Displayed in the Hex Editor and Importing and Exporting Memory Files segments. Added the Edit value menu. • Added Example: Using the In-System Memory Content Editor with the SignalTap II Embedded Logic Analyzer.	—
Aug. 2004 v1.1	Minor typographical corrections.	—
June 2004 v1.0	Initial release.	—

16. Design Debugging Using In-System Sources and Probes

QII53021-7.2.0

Introduction

Traditional debugging techniques often involve using an external pattern generator to exercise the logic and a logic analyzer to study the output waveforms during run-time. The SignalTap® II Logic Analyzer and SignalProbe allow you to read or "tap" internal logic signals during run-time as a way to debug your logic on-chip. While this is useful, the debugging cycle efficiency may be enhanced with the ability to drive any internal signal manually within your design. By doing this you can perform the following activities:

- Force the occurrence of trigger conditions setup in the SignalTap II Logic Analyzer
- Create simple test vectors to exercise your design without using external test equipment
- Dynamically control run-time control signals with the JTAG chain

With the introduction of the In-System Sources and Probes feature in the Quartus® II software beginning with version 7.1, Altera extends the portfolio of verification tools. The In-System Sources and Probes feature allows you to easily control any internal signal, providing you with a completely dynamic debugging environment. Coupled with either the SignalTap II Logic Analyzer or SignalProbe, the In-System Sources and Probes feature gives you a powerful debugging environment in which to generate stimuli and solicit responses from your logic design.

This chapter addresses the following topics:

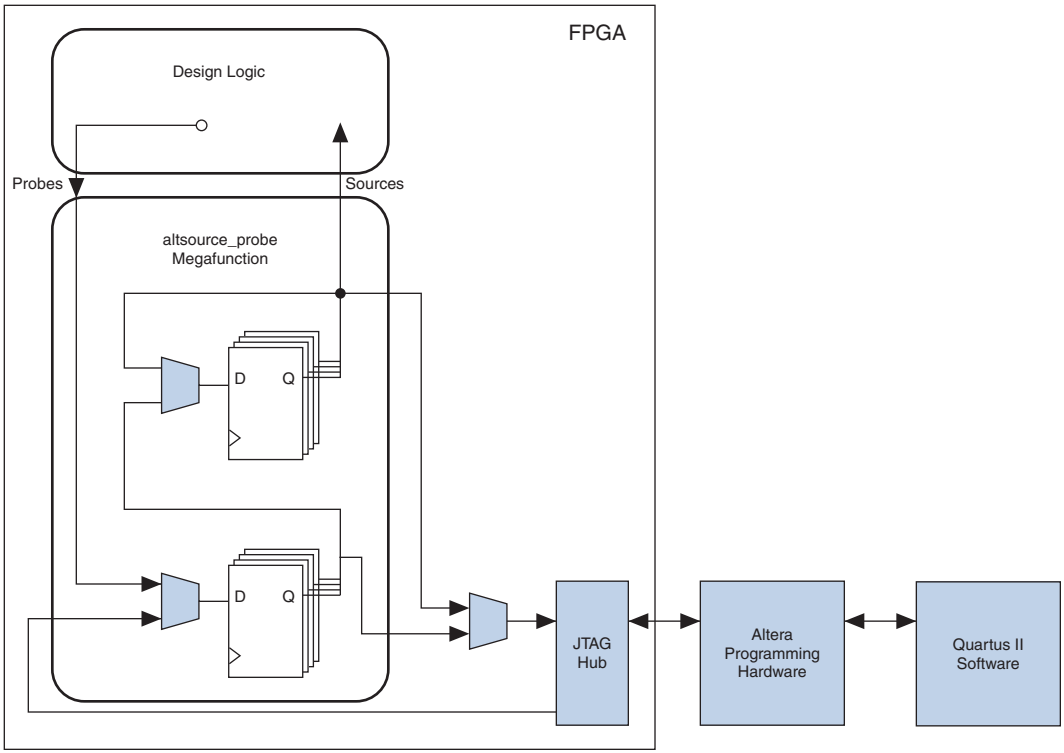
- "Design Flow Using In-System Sources and Probes" on page 16–4
- "Running the In-System Sources and Probes Editor" on page 16–9
- "TCL Support" on page 16–14
- "Design Example: Dynamic PLL Reconfiguration" on page 16–18

Overview

The In-System Sources and Probes feature consists of the altsource_probe megafunction and an interface to control the altsource_probe megafunction instances during run-time. Each altsource_probe megafunction instance provides you with source output ports and probe input ports, where source ports drive selected signals and probe ports sample selected signals. Upon compilation, the altsource_probe megafunction sets up a register chain to either drive or sample the selected nodes in your logic design. During runtime, the In-System Sources and Probes interface uses a JTAG connection to shift data to and

from the altsource_probe megafunction instances. Figure 16–1 shows a block diagram of the components that make up the In-System Sources and Probes feature.

Figure 16–1. In-System Sources and Probes Block Diagram



The altsource_probe megafunction hides the detailed transactions between the JTAG Hub and the registers instrumented in your design to give you a basic building block for stimulating and probing your design. Moreover, the In-System Sources and Probes feature provides single-cycle samples and single-cycle writes to the selected logic nodes. This provides an easy way to input simple virtual stimuli and an easy way to capture the current value on instrumented nodes. Because In-System Sources and Probes gives you access to logic nodes within your design, this feature can be used during the debugging process to toggle the inputs of low-level components. If used in conjunction with the SignalTap II Logic Analyzer, you can force trigger conditions to help isolate your problem and shorten your debugging process.

Additionally, the ease of use of the In-System Sources and Probes feature makes it ideal for implementing control signals as virtual stimuli. This feature can be especially helpful during for prototyping your design. Examples of such applications could include the ability to do the following:

- Create virtual push buttons
- Create a virtual front panel to interface with your design
- Mimic external sensor data
- Monitor and change run-time constants on the fly

In-System Sources and Probes supports Tcl commands to interface with all your altsource_probe instances to increase the level of automation.



The Virtual JTAG Megafunction and the In-System Memory Content Editor also give you the capability to drive virtual inputs into your design. The Virtual JTAG Megafunction gives you a greater level of control (compared to In-system Sources and Probes) in how your design communicates with the JTAG Hub at the cost of greater complexity. With the Virtual JTAG megafunction, you can design your own customized register scan chain to drive and control your logic through the JTAG port. The In-System Memory Content Editor is used specifically for reading and writing memory contents at runtime.



For more details about the Virtual JTAG Megafunction, refer to the *sld_virtual_jtag Megafunction User Guide*. For information about the In-System Memory Content Editor, refer to the *In-System Updating of Memory and Constants* chapter in volume 3 of the *Quartus II Handbook*.

Hardware and Software Requirements

The following components are required to use In-System Sources and Probes:

- Quartus II design software
  or
- Quartus II Web Edition (with TalkBack feature enabled)

- Download Cable (USB-Blaster™ download cable or ByteBlaster™ cable)
- Altera® development kit or user design board with JTAG connection to device under test

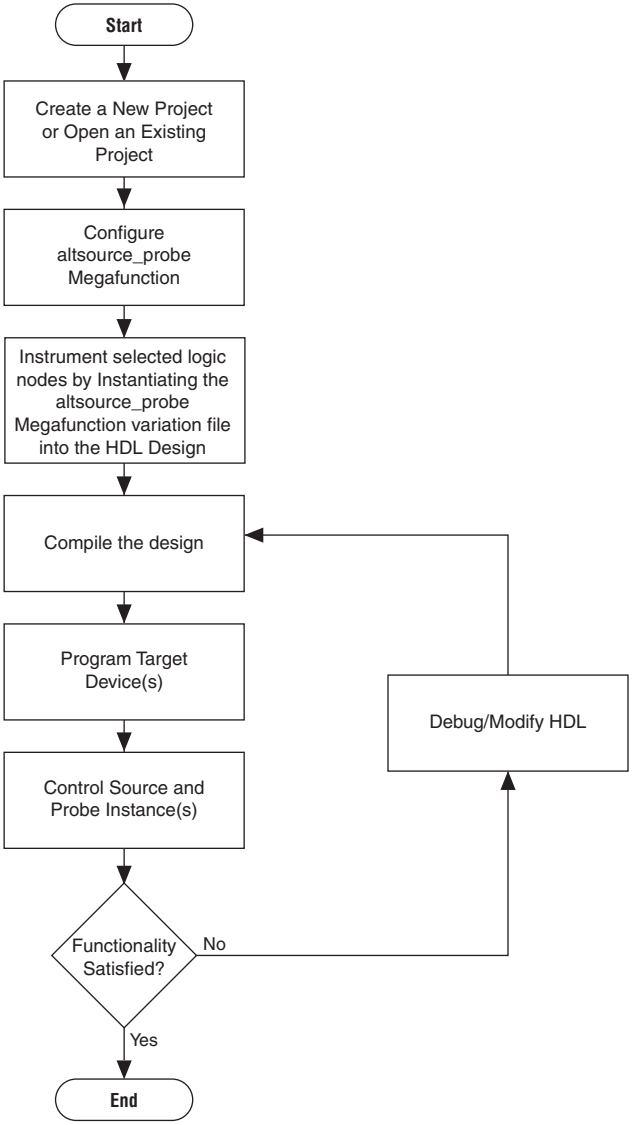
The In-System Sources and Probes feature supports the following device families:

- Arria™ GX
- Stratix® III
- Stratix II
- Stratix II GX
- Stratix
- Stratix GX
- HardCopy® II
- HardCopy Stratix
- Cyclone® III
- Cyclone® II
- Cyclone
- MAX® II
- APEX™ II
- APEX 20KE
- APEX 20KC
- APEX 20K

Design Flow Using In-System Sources and Probes

In-System Sources and Probes supports an RTL flow in which your design nodes are instrumented in your HDL code via instantiation of the altsource_probe megafunction. After your device is compiled with the design nodes that you want instrumented, you can control your altsource_probe instances via the Sources and Probes Editor GUI or via a Tcl interface. The complete design flow is shown in Figure 16–2.

Figure 16–2. FPGA Design Flow Using In-System Sources and Probes



Configuring the altsource_probes Megafunction

To add in-system sources and probes functionality to your design, you must first instantiate the altsource_probe megafunction variation file. The altsource_probe megafunction can be easily configured using the MegaWizard® Plug-In Manager. Each source or probe port can be up to 256 bits wide. You can have up to 128 instances of the altsource_probe megafunction in your design. The following steps will guide you through the steps necessary to configure the altsource_probe megafunction:

1. On the Tools menu, click **MegaWizard Plug-In Manager**.
2. Select **Create a new custom megafunction variation**.
3. Click **Next**.
4. On Page 2a, make the following selections:
    a. In the Installed Plug-Ins list, expand the JTAG-accessible Extensions folder. In the JTAG-accessible Extensions list, select **In-System Sources and Probes**.
    b. Make sure that the currently selected device family matches the device you are targeting.
    c. Select an output file type and enter the desired name of the altsource_probe megafunction. You can choose AHDL (.tdf), VHDL (.vhd), or Verilog HDL (.v) as the output file type.
5. Click **Next**.
6. On Page 3, make the following selections:
    a. Make sure that the currently selected device family matches the device that you are targeting.
    b. Under **Do you want to specify an Instance Index?**, turn on **Yes**.
    c. Specify the Instance ID of this instance.
    d. Specify the width of the probe port. The width can be from 1 bit to 256 bits wide.
    e. Specify the width of the source port. The width can be from 1 bit to 256 bits wide.

7. On Page 3, you can click **Advanced Options** and specify other parameters. The following options are included:

   - **What is the initial value of the source port, in hexadecimal?** This option allows you to specify the initial value driven on the source port at run-time.
   - **Write data to the source port synchronously to the source clock.** This allows you to synchronize your source port write transactions with the clock domain of your choice.
   - **Create an enable signal for the registered source port.** When enabled, this creates a clock enable input for the synchronization registers. This option is enabled only when the **Write data to the source port synchronously to the source clock** option is enabled.

Table 16–1 summarizes the configurable fields for the altsource_probe megafunction.

Table 16–1. MegaWizard Plug-In Manager —altsource_probe (page 3) Options

Options	Description
Currently selected device family	Specifies the device family.
Do you want to specify an Instance Index?	Specifies the numeric index of the megafunction instance during run-time (from 0 to 127).
The 'Instance ID' of this Instance (optional):	Specifies the four character ID tag of the megafunction in the instance manager window of the Sources and Probes Editor.
How wide should the probe port be?	Specifies the number of signals to be read by In-System Sources and Probes.
How wide should the source port be?	Specifies the number of signals to be driven by In-System Sources and Probes.
What is the initial value of the source port (under **Advanced Options**)	Specifies the initial value driven on the source port at run time.
Write data to the source port synchronously to the source clock. Each bit in the source port will utilize two additional registers to achieve metastability (under **Advanced Options**)	Turning on this option allows you to synchronize your source port write transactions with the clock domain of your choice.
Create an enable signal for the registered source port (configured under **Advanced Options**)	Turning on this option creates a clock enable input for the synchronization registers.

Instantiating the altsource_probe Megafunction

The MegaWizard Plug-in Manager produces the necessary variation file and the instantiation template based on your inputs to the MegaWizard. Use the template to instantiate the altsource_probe megafunction variation file in your design. The port information is shown in Table 16–2.

Table 16–2. **altsource_probe Megafunction Port Information**

Port Name	Required?	Direction	Comments
`Probe[]`	No	Input	The outputs from the user's design.
`Source_clk`	No	Input	Source Data is written synchronously to this clock. This input is required if the Source Clock option is turned on in the Advanced Options box in the MegaWizard Plug-in Manager.
`Source_ena`	No	Input	Clock enable signal for `source_clk`. This input is required if specified in the Advanced Options box in the MegaWizard Plug-in Manager.
`Source[]`	No	Output	Used to drive inputs to user design.

You can include up to 128 instances of the altsource_probe megafunction in your design, provided that there are available logic resources in your device. Each instance of the altsource_probe megafunction uses a pair of registers per signal for the width of the widest port it contains. Additionally, there will be some fixed overhead logic to accommodate communication between the altsource_probe instances and the JTAG controller. An additional pair of registers per source port is added for synchronization if it is specified.

Compiling the Design

When you compile your design with the In-System Sources and Probes megafunction instantiated, an instance of the altsource_probe instance and sld_hub megafunctions are added to your compilation hierarchy automatically. These two instances allow communication between the JTAG controller and your instrumented logic.

To modify your In-System Sources and Probes connections, you can modify the number of connections to your design by editing the altsource_probe megafunction. You can open the MegaWizard Plug-In Manager for the design instance you want to modify by double-clicking the desired instance in the Project Navigator. You can then modify the connections in the HDL source file. You must recompile your design when you are finished editing it.

Because the design cycle is iterative in nature, you can use the Quartus II incremental compilation feature to reduce compilation time. Incremental compilation allows you to organize your design into logical partitions. During recompilation of a design, incremental compilation preserves the compilation results and performance of unchanged partitions and reduces design iteration time by compiling only modified design partitions.



For more information about Incremental Compilation, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

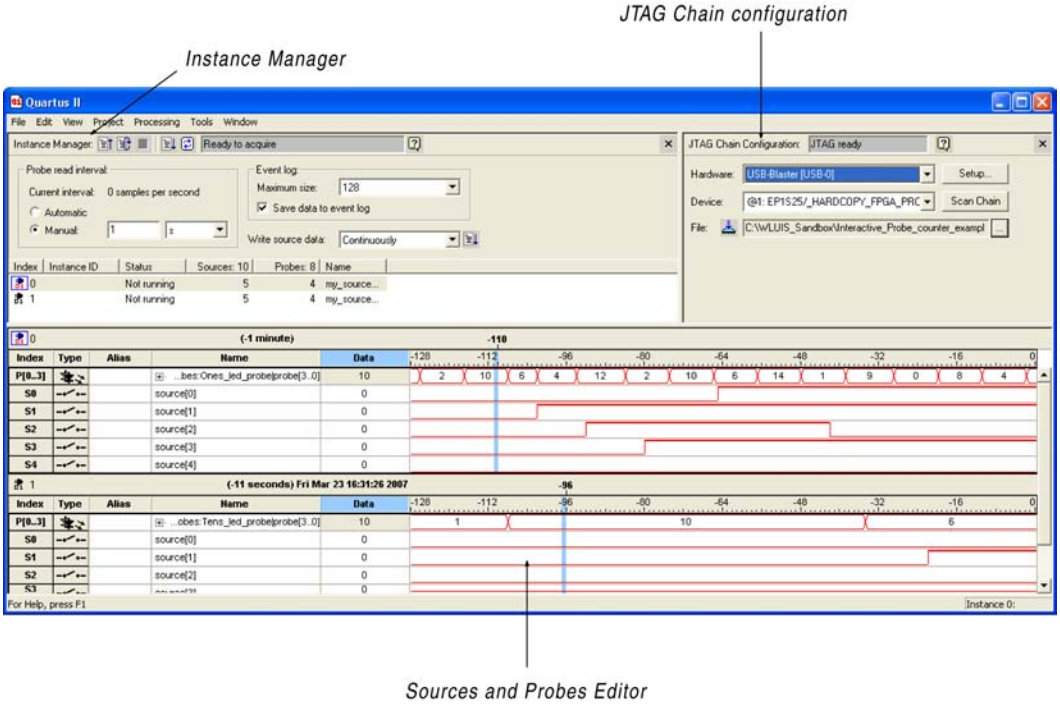
Running the In-System Sources and Probes Editor

The In-System Sources and Probes Editor is a GUI that gives you control over all of the altsource_probe megafunction instances within your design. It displays all available run-time controllable instances of the altsource_probe megafunction in your design, provides a push-button interface to drive all of your source nodes, and a logging feature to store your probe and source data.

To run the In-System Sources and Probes Editor, from the **Tools** menu, click **In-System Sources and Probes Editor.**

Figure 16–3 shows the Editor window.

Figure 16–3. In-System Sources and Probes Editor



The In-System Sources and Probes Editor is made up of three panes:

- **JTAG Chain configuration**—Allows you to specify programming hardware, device, and file settings that the In-System Sources and Probes Editor uses to program and acquire data from a device.
- **Instance Manager**—Displays information about the instances generated when you compile a design, and allows you to control the data the In-System Sources and Probes Editor acquires.
- **Sources and Probes Editor**—Logs all the data read from the selected instance and allows you to modify source data to be written to your device.

Using the In-System Sources and Probes Editor does not require you to open a Quartus II project. The In-System Sources and Probes Editor retrieves all instances of the altsource_probe megafunction by scanning the JTAG chain and sending a query to the specific device selected in the JTAG Chain Configuration pane. Also, you can use a previously saved configuration to run the In-System Sources and Probes Editor.

Each In-System Sources and Probes Editor window can access the altsource_probe megafunction instances in a single device. If you have more than one device containing megafunction instances in a JTAG chain, you can launch multiple In-System Sources and Probes Editor windows to access the megafunction instances in each of the devices.

Programming Your Device Using the JTAG Chain Configuration Window

After compilation is complete, you must configure your FPGA before using In-System Sources and Probes. To configure a device for use with the In-System Sources and Probes, perform the following steps:

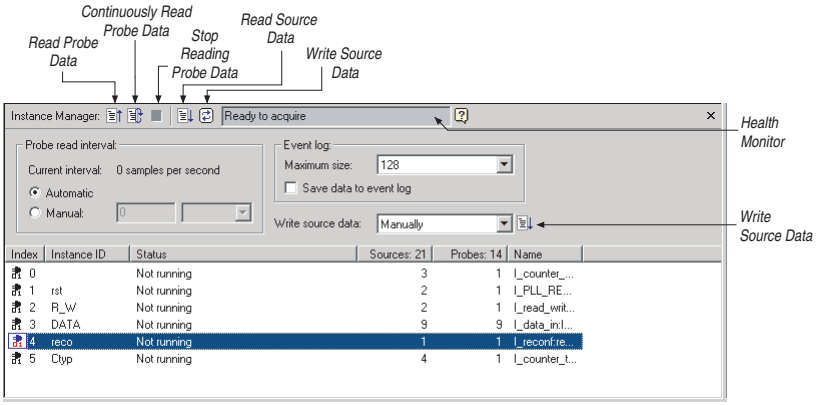
1. Open the In-System Sources and Probes Editor.
2. Under JTAG Chain Configuration, point to **Hardware** and select the desired hardware communications device. You may be prompted to configure your hardware; in this case, click **Setup**.
3. From the **Device** list, select the FPGA device to which you want to download the design (it may be automatically detected). You may have to click **Scan Chain** to detect your target device.
4. In the JTAG Configuration window, click **Browse** and select the SRAM Object File (**.sof**) that includes the In-System Sources and Probes instance or instances. (Note that it may be automatically detected).
5. Click **Program Device** (next to **File:**) to program the target device.

Instance Manager

The Instance Manager provides a list of all altsource_probe instances in the design and allows you to configure how data is acquired from or written to those instances.

The Instance Manager is shown in Figure 16–4.

Figure 16–4. Instance Manager



The following buttons and sub-panes are provided in the Instance Manager:

- **Read Probe Data**—Samples the probe data in the selected instance and displays it in the Sources and Probes Editor Window
- **Continuously Read Probe Data**—Continuously samples the probe data of the selected instance and displays it in the Sources and Probes Editor Window; you can modify the sample rate via the Probe read interval setting
- **Stop Continuously Reading Probe Data**—Cancels continuous sampling of probe of selected instance
- **Write Source Data** sub-pane —Writes data to all source nodes of the selected instance
- **Probe Read Interval** sub-pane—Displays the sample interval of all the In-system Sources and Probe instances in your design; you can modify the sample interval by clicking **Manual**
- **Event Log** sub-pane—controls the event log in the Sources and Probes Editor Window
- **Write Source Data** sub-pane—Allows you to manually or continuously write data to the system

The status of each instance is also displayed beside each entry in the Instance Manager. The status indicates if the instance is **Not running Offloading data**, **Updating data**, or if an "Unexpected JTAG communication error" occurs. The health monitor provides useful information about the status of the editor.

Sources and Probes Editor Window

The Sources and Probes Editor window organizes and displays the data from all sources and probes in your design, organized according to the index number of the instance. The editor provides an easy way to manage your signals, allowing you to rename signals or to group them into buses. All data collected from source and probe nodes is recorded in the event log and displayed as a timing diagram.

Reading Probe Data

You can read data by selecting the desired altsource_probe instance in the Instance Manager and clicking **Read Probe Data**. This produces a single sample of the probe data and updates the data column of the selected index in the Sources and Probes Editor window. You can save the data to an event log by turning on the **Save data to event log** option in the Instance Manager.

If you want to sample data from your probe instance continuously, in the Instance Manager, click the instance you want to read, and then click **Continuously read probe data**. While reading, the status of the active instance will show **Unloading**. You can read continuously from multiple instances.

You can access read data by using the right-click menus in the Instance Manager.

To adjust the probe read interval, in the Instance Manager, turn on the **Manual** option in the **Probe read interval** sub-pane, and specify the desired sample rate in the text field next to the **Manual** option. The maximum sample rate depends on your computer setup. The actual sample rate is shown in the **Current interval** box. The event log window buffer size can be adjusted in the **Maximum Size** box.

Writing Data

To modify the source data to be written into the altsource_probe instance, click in the name field of the signal you want to change. For buses of signals, you can double-click on the data field and type in the value to be driven out to the altsource_probe instance. The In-System Sources and Probes Editor stores the modified source data values into a temporary

buffer. Modified values that have not been written out to the altsource_probe instances appear in red. To update the altsource_probe instance, highlight the instance in the Instance Manager and click **Write source data**. The **Write source data** function is also available via the shortcut menus in the Instance Manager.

You can choose to have the values stored in the In-System Sources and Probes Editor continuously update the altsource_probe instances. By doing so, any modifications you make to the source data buffer are written immediately to the altsource_probe instances. To continuously update the altsource_probe instances, change the **Write source data** field from **Manually** to **Continuously**.

Data Organization

The main editor window allows you to group signals into buses, and allows you to modify the display options of the data buffer.

To create a group of signals, select the node names you want to group, right-click and select **Group**. You can modify the display format in the Bus Display Format and the Bus Bit order submenus.

The Sources and Probes Editor Window allows you to rename any signal. To rename a signal, double-click the name of the desired signal and type in the new name.

The event log contains a record of the most recent samples. The buffer size is adjustable, up to 128k samples. The time stamp for each sample is logged and is displayed above the event log of the instance being examined as you move your mouse pointer over the data samples.

You can save the changes that you have made and the recorded data into a Sources and Probes File (**.spf**). To save changes, on the File menu, click **Save**. The file contains all of the modifications you made to the signal groups, as well as the current data event log.

TCL Support

To support automation, In-system Sources and Probes supports the procedures described in this chapter in the form of Tcl commands. The Tcl package for In-System Sources and Probes is included by default when you run **quartus_stp**.

The Tcl interface for In-System Sources and Probes provides a powerful platform to help you debug your design. It is especially helpful for debugging designs that require toggling multiple sets of control inputs. You can aggregate multiple commands using a Tcl script to define your own custom command set.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Reference Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Table 16–3 shows the Tcl command you can use with In-System Sources and Probes.

Table 16–3. In-System Sources and Probes Tcl Commands

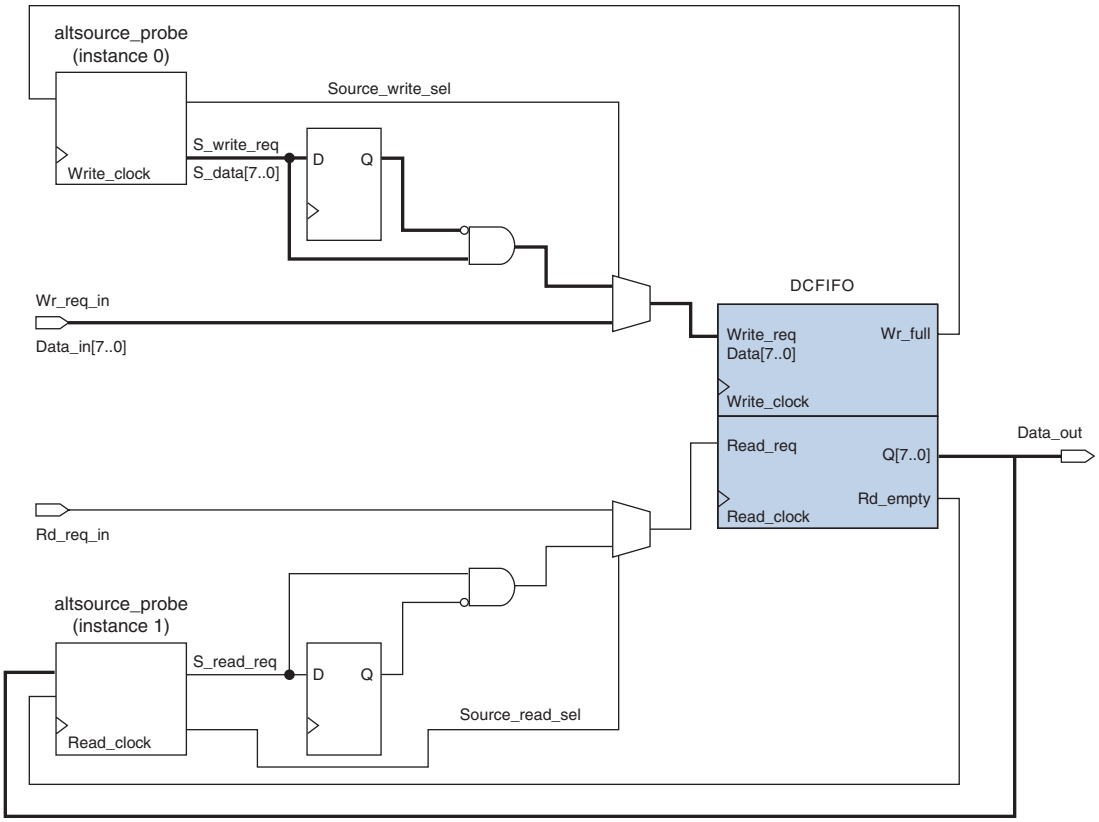
Command	Argument	Description
`start_insystem_source_probe`	`-device_name` <*device name*> `-hardware_name` <*hardware name*>	Opens a handle to a device using the specified hardware. Call this command before starting any transactions.
`get_insystem_source_probe_instance_info`	`-device_name` <*device name*> `-hardware_name` <*hardware name*>	Returns a list of all `altsource_probe` instances in your design. Each record returned will be in the following format: {<*instance Index*>, <*source width*>, <*probe width*>, <*instance name*>}
`read_probe_data`	`-instance_index` <*instance_index*> `-value_in_hex` *(optional)*	Retrieves the current value of the probe. A string is returned specifying the status of each probe, with the MSB as the left-most bit.
`read_source_data`	`-instance_index` <*instance_index*> `-value_in_hex` *(optional)*	Retrieves the current value of the sources. A string is returned specifying the status of each source, with the MSB as the left-most bit.
write_source_data	`-instance_index` <*instance_index*> `-value` <*value*> `-value_in_hex` *(optional)*	Sets the value of the sources. A binary string is sent to the source ports, with the MSB as the left-most bit.
end_interactive_probe	None	Releases the JTAG chain. Issue this command when all transactions are finished.

Example 16–1 shows an excerpt from a Tcl script with procedures that control the altsource_probe instances of the design as shown in Figure 16–5. The example design contains a DCFIFO with altsource_probe instances to read from and to write to the DCFIFO. A set of control muxes are added into the design to control the flow of data to

the DCFIFO between the input pins and the altsource_probe instances. A pulse generator is added to the read request and write request control lines to guarantee a single sample read or write. The altsource_probe instances, when used with the script in Example 16–1, provides visibility into the contents of the FIFO by performing single sample write and read operations and reporting the state of the full and empty status flags.

The Tcl script can be useful in debugging situations where you may want to either empty or preload the FIFO in your design. As an example, you can use this feature to preload the FIFO to match a trigger condition you have set up within the Signal Tap II Logic Analyzer.

Figure 16–5. A DCFIFO Example Design Controlled by the Tcl Script in Example 16–1



Example 16–1. Tcl Script Procedures for Reading and Writing to the DCFIFO in Figure 16–5

```
## Setup USB hardware  - assumes only USB Blaster is installed and
## an FPGA is the only device in the JTAG chain

set usb [lindex [get_hardware_names] 0]
set device_name [lindex [get_device_names -hardware_name $usb] 0]
## write procedure :  argument value is integer

proc write {value} {

    global device_name usb
    variable full

    start_insystem_source_probe -device_name $device_name -hardware_name \
    $usb

    #read full flag
    set full [read_probe_data -instance_index 0]


    if {$full == 1} {end_insystem_source_probe
    return "Write Buffer Full"
    }


    ##toggle select line, drive value onto port, toggle enable
    ##bits 7:0 of instance 0 is S_data[7:0]; bit 8 = S_write_req;
    ##bit 9 = Source_write_sel

    ##int2bits is custom procedure that returns a bitstring from an integer
    ## argument


    write_source_data -instance_index 0 -value /[int2bits [expr 0x200 | \
    $value]]
    write_source_data -instance_index 0 -value [int2bits [expr 0x300 | \
    $value]]

    ##clear transaction

    write_source_data -instance_index 0 -value 0

    end_insystem_source_probe
}

proc read {} {

    global device_name usb
    variable empty
    start_insystem_source_probe -device_name $device_name -hardware_name \
    $usb
```

```
    ##read empty flag : probe port[7:0] reads FIFO output; bit 8 reads
    ##empty_flag

    set empty [read_probe_data -instance_index 1]


    if {[regexp {1........} $empty]} { end_insystem_source_probe
    return "FIFO empty" }



    ## toggle select line for read transaction
    ## Source_read_sel = bit 0; s_read_reg = bit 1

    ## pulse read enable on DC FIFO
    write_source_data -instance_index 1 -value 0x1 -value_in_hex
    write_source_data -instance_index 1 -value 0x3 -value_in_hex

    set x [read_probe_data -instance_index 1 ]

    end_insystem_source_probe

    return $x
}
```

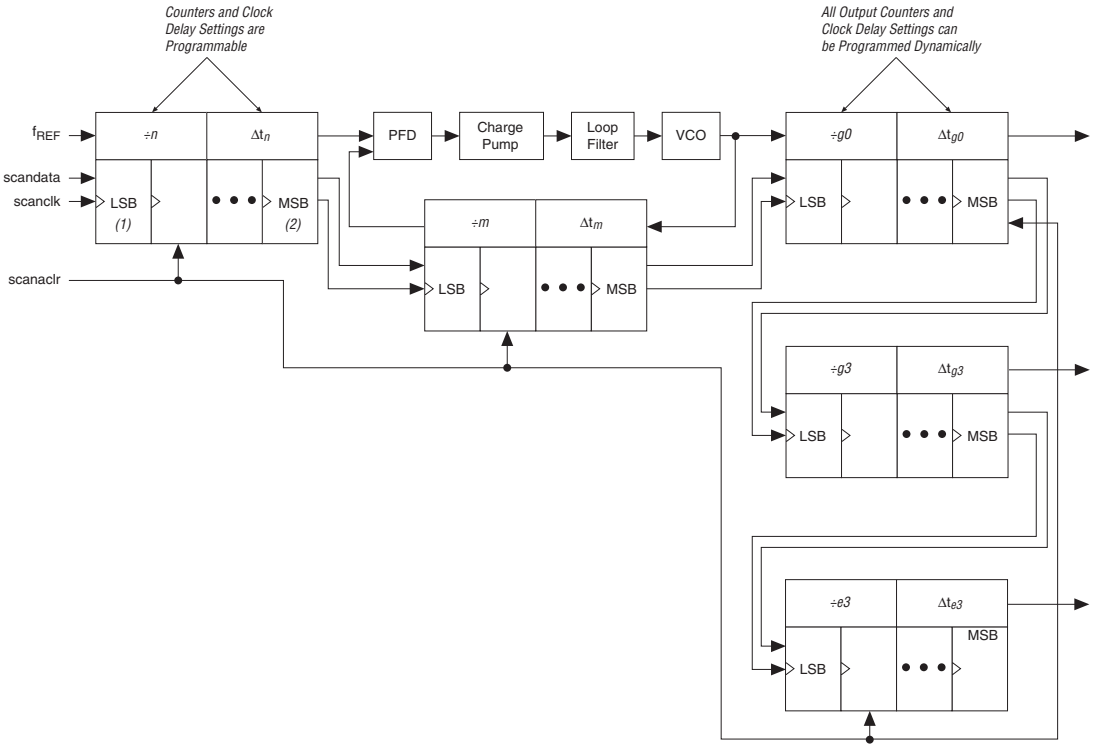
Design Example: Dynamic PLL Reconfiguration

The ease of use of the In-System Sources and Probes feature can be extremely helpful in creating a virtual front panel during the prototyping phase of your design. Relatively simple designs of high functionality can be created in a short amount of time. The following PLL reconfiguration example demonstrates how the In-System Sources and Probes feature is used to provide a GUI to dynamically reconfigure a Stratix PLL.

Stratix PLLs allows you to dynamically update PLL coefficients during run-time. Each enhanced PLL within the Stratix device contains a register chain that allows you to modify the pre-scale counters (*m* and *n* values), output divide counters, and delay counters. In addition, the altpll_reconfig megafunction provides an easy interface to access the register chain counters. The altpll_reconfig megafunction provides a cache containing all modifiable PLL parameters. After you have updated all of the PLL parameters in the cache, the alt_pll_reconfig megafunction drives the PLL register chain to update the PLL with the updated parameters. Figure 16–6 shows a Stratix enhanced PLL with reconfigurable coefficients.

Stratix II and Stratix III devices also allow you to dynamically reconfigure PLL parameters. For more information about these families, refer to the appropriate data sheet. For more information about dynamic PLL reconfiguration, refer to *AN 282: Implementing PLL Reconfiguration in Stratix & Stratix GX Devices* or *AN 367: Implementing PLL Reconfiguration in Stratix II Devices*.

Figure 16–6. Stratix-Enhanced PLL with Reconfigurable Coefficients



The following design example uses an altsource_probe instance to update the PLL parameters in the altpll_reconfig megafuntion cache. The altpll_reconfig megafunction connects to an enhanced PLL in a Stratix FPGA to drive the register chain containing the PLL reconfigurable coefficients. This design example uses a Tcl/Tk script to generate a GUI where you can enter in new *m* and *n* values for the enhanced PLL. The Tcl script extracts the m and n values from the GUI, shifts the values out to the altsource_probe instances to update the values in the altpll_reconfig megafunction cache and asserts the reconfig signal on the altpll_reconfig megafunction. The reconfig signal on the altpll_reconfig megafunction

starts the register chain transaction to update all PLL reconfigurable coefficients. A block diagram of design example is shown in Figure 16–7. The Tk GUI is shown in Figure 16–8.

Figure 16–7. Block Diagram of Dynamic PLL Reconfiguration Design Example

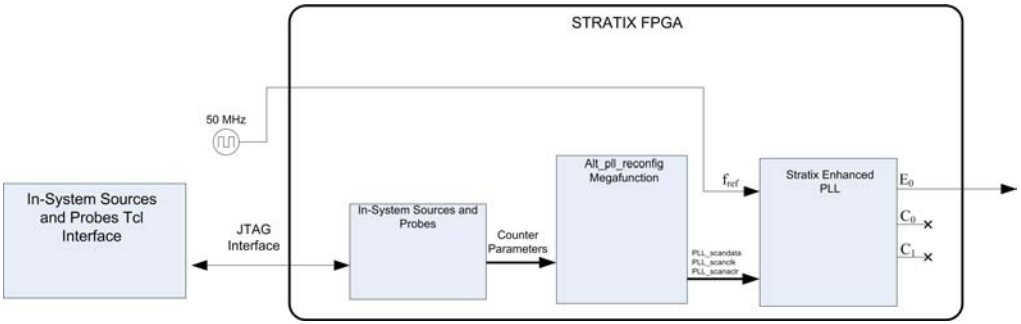
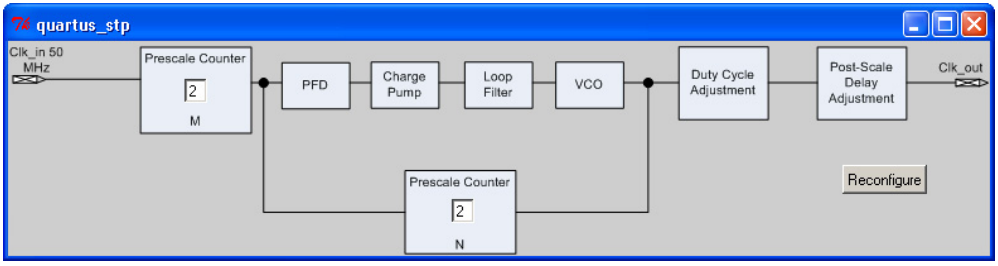


Figure 16–8. Interactive PLL Reconfiguration GUI Created with Tk and In-System Sources and Probes Tcl Package



This design example was created using a NIOS® Development Kit, Stratix Edition. The file **sourceprobe_DE_dynamic_pll.zip** contains all of the necessary files for running this design example:

- **Readme.txt**—A text file that describes the files contained in the Design Example and provides instructions on running the Tk GUI shown in Figure 16–8.
- **Interactive_Reconfig.qar**—The archived Quartus II project for this Design Example

You can download the **sourceprobe_DE_dynamic_pll.zip** file in the *Quartus II Handbook* volume 3 section of the Altera Literature web page.

Conclusion

In-System Sources and Probes can provide stimuli and get responses from the target design during run-time. With its simple and intuitive interface, you can provide virtual inputs into your design during run-time without using external equipment. When used in conjunction with SignalTap II, you can use In-System Sources and Probes to provide greater control of the signals in your design, and thus help shorten the verification cycle. Also, with its ability to create virtual inputs into your design, you can create simple, yet powerful applications to interact with your design.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *sld_virtual_jtag Megafunction User Guide*
- *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 16–4 shows the revision history for this chapter.

Table 16–4. Document Revision History

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 16–21.	—
May 2007 v7.1.0	Initial Release.	—

Section VI. Formal Verification

The Quartus® II software easily interfaces with EDA formal design verification tools such as the Cadence Incisive Conformal and Synplicity Synplify software. In addition, the Quartus II software has built-in support for verifying the logical equivalence between the synthesized netlist from Synplicity Synplify and the post-fit Verilog Quartus Mapped (**.vqm**) files using Incisive Conformal software.

This section discusses formal verification, how to set-up the Quartus II software to generate the VQM file and Incisive Conformal script, and how to compare designs using Incisive Conformal software.

This section includes the following chapters:

- Chapter 17, Cadence Encounter Conformal Support
- Chapter 18, Synopsys Formality Support



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

17. Cadence Encounter Conformal Support

QII53011-7.2.0

Introduction

The Quartus® II software provides formal verification support for Altera® designs through interfaces with a formal verification EDA tool, the Cadence Encounter Conformal software.

Use the Encounter Conformal software to verify the functional equivalence of a post-synthesis Verilog Quartus Mapping netlist from the Synplicity Synplify Pro software and the post-fit Verilog Output File from the Quartus II software. You can also use the Encounter Conformal software to verify the functional equivalence of the register transfer level (RTL) source code and post-fit Verilog Output File from the Quartus II software when using Quartus II integrated synthesis. These formal verification flows support designs for the Cyclone®, Cyclone II, Stratix®, Stratix II, Stratix GX, Stratix II GX, Stratix III, Arria™ GX, and HardCopy® II device families.

There are two types of formal verification—equivalence checking and model checking. This chapter discusses equivalence checking using the Cadence Encounter Conformal software.

This chapter contains the following sections:

- "Formal Verification Design Flow" on page 17–2
- "RTL Coding Guidelines for Quartus II Integrated Synthesis" on page 17–5
- "Generating the Post-Fit Netlist Output File and the Encounter Conformal Setup Files" on page 17–10
- "Understanding the Formal Verification Scripts for Encounter Conformal" on page 17–18
- "Comparing Designs Using Encounter Conformal" on page 17–21
- "Known Issues and Limitations" on page 17–24
- "Black Box Models" on page 17–28
- "Conformal Dofile/Script Example" on page 17–30

Equivalence checking uses mathematical techniques to compare the logical equivalence of the two versions of the same design rather than using test vectors to perform simulation. The two compared versions could be post-map design and post-fit design, or RTL design and post-fit design. Equivalence checking greatly shortens the verification cycle of the design.

Formal Verification Versus Simulation

Formal verification cannot be considered as a replacement to the vector-based simulation. Formal verification only complements the existing vector-based simulation techniques to speed up the verification cycle. Vector-based simulation techniques of gate level designs can take a considerable amount of time.

Vector-based simulation techniques can be used to do the following:

- Verify design functionality
- Verify timing specifications
- Debug designs

Formal Verification: What You Need to Know

If you use formal verification techniques to verify logic equivalence of your design, you can save time by forgoing a comprehensive vector-based simulation of the gate level design. However, there may be impact on area and performance during recompilation of your design with the Quartus II software if you have chosen to use formal verification flow for Cadence Conformal LEC software. The area and performance of your design may be affected by the following factors:

- Hierarchy preservation
- ROM implementation by logic elements (LEs)
- Retiming is disabled

Refer to "Known Issues and Limitations" on page 17–24 before you consider using the formal verification flow in your design methodology.

Formal Verification Design Flow

Altera supports formal verification using the Encounter Conformal software for the following two synthesis tools:

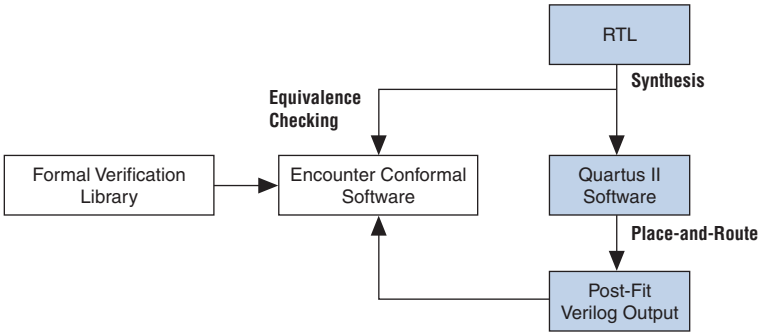
- Quartus II Integrated Synthesis
- Synplify Pro

The following sections describe the supported design flows for these synthesis tools.

Quartus II Integrated Synthesis

The design flow for formal verification using the Quartus II integrated synthesis is shown in Figure 17–1. This flow performs equivalency checking for the RTL source code and the post-fit netlist generated by the Quartus II software. The RTL source code can be in Verilog or VHDL format. The post-fit netlist generated by the Quartus II software is always in Verilog format.

Figure 17–1. Formal Verification Using Quartus II Integrated Synthesis and the Encounter Conformal Software



EDA Tool Support for Quartus II Integrated Synthesis

The formal verification flow using the Quartus II software and Cadence Encounter Conformal software supports the following software versions and operating systems:

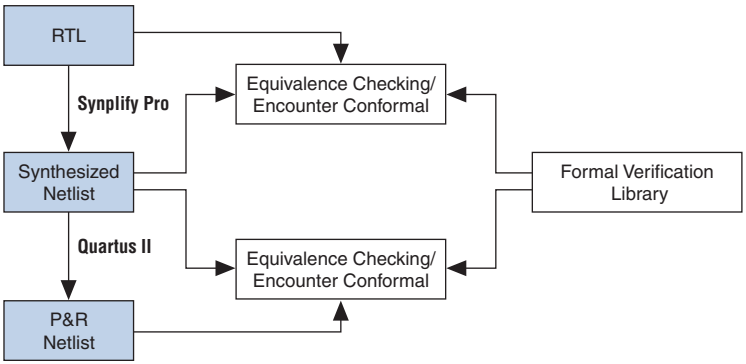
- Quartus II software beginning with version 4.2
- Cadence Encounter Conformal software beginning with 4.3.5A
- Solaris and Linux operating systems

Synplify Pro

The design flow for formal verification using Synplify Pro Synthesis performs equivalency checking for the post-synthesis netlist from Synplify Pro and the post fit netlist generated by Quartus II software, as shown in Figure 17–2.

For additional information about performing equivalency checking between RTL and post-synthesis netlist generated from Synplify Pro software, refer to the Synplify Pro documentation.

Figure 17–2. Formal Verification Flow Using Synplify Pro and the Encounter Conformal Software



EDA Tool Support for Synplify Pro

The formal verification flow using the Quartus II software, the Synplicity Synplify Pro, and the Cadence Encounter Conformal software supports the software versions and operating systems shown in Table 17–1.

Table 17–1. Compatible Software Versions

Quartus II Software Version	Cadence Conformal LEC Version	Synplify Pro Version
4.1	4.3.0.a	7.6.1
4.2	4.3.5.a	8.0
5.0	5.1	8.1
5.1	5.1	8.4
6.0	5.2	8.5
6.1	6.1	8.6.2
7.0	6.1	8.6.2
7.1	6.2	8.8.1
7.2	7.1	9.0

RTL Coding Guidelines for Quartus II Integrated Synthesis

The Cadence Encounter Conformal software can compare the RTL code against the post-fit netlist generated by the Quartus II software. The Encounter Conformal software and the Quartus II integrated synthesis parse and compile the RTL description in slightly different ways. The Quartus II software supports some RTL features that the Encounter Conformal software does not support, and vice versa. The style of the RTL code is of particular concern because neither tool supports some constructs, leading to potential formal verification mismatches; for example, state machine extraction, wherein different encoding mechanisms can result in different structures. Therefore, for successful verification, both tools must interpret the RTL code in the same manner.

The following section provides information on recognizing and preventing problems that can arise in the formal verification flow.



For more details about RTL coding styles for Quartus II Integrated Synthesis, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.



Some of the coding guidelines apply to both Quartus II Integrated Synthesis and Synplify Pro flow, as indicated in each of the guidelines in the following sections.

Synthesis Directives and Attributes

Synthesis directives, also known as pragmas, play an important role in successful verification of RTL against the post-fit Verilog Output netlist from the Quartus II software.

Pragmas and trigger keywords that are supported in Quartus II integrated synthesis and Encounter Conformal are also supported in the formal verification flow. The Quartus II integrated synthesis and Encounter Conformal both support the trigger keywords **synthesis** and **synopsys**. When the Quartus II software does not recognize a keyword such as **verplex**, the keyword is disabled in the formal verification scripts produced for use with the Cadence Conformal software. Therefore, it is important to use caution with unsupported pragmas because they can lead to verification mismatches.

For example, you can use the Quartus II integrated synthesis to synthesize RTL code with the synthesis directive `read_comments_as_HDL`.

Example 17–1. Verilog HDL Example of Read Comments as HDL

```
// synthesis read_comments_as_HDL on
// my_rom lpm_rom (.address (address),
// .data (data));
// synthesis read_comments_as_HDL off
```

Example 17–2. VHDL Example of Read Comments as HDL

```
-- synthesis read_comments_as_HDL on
-- my_rom : entity lpm_rom
-- port map (
-- address => address,
-- data => data, );
-- synthesis read_comments_as_HDL off
```



The Encounter Conformal software does not support the synthesis directive `read_comments_as_HDL`, and the directive has no affect on the Encounter Conformal software.

Table 17–2 lists supported pragmas and trigger keywords for formal verification.

Table 17–2. Supported Pragmas and Trigger Keywords for Formal Verification

Pragmas *(1)*	Trigger Keywords
full_case parallel_case pragma synthesis_off synthesis_on translate_off translate_on	synthesis synopsys

Note to Table 17–2:

(1) Do not use Verilog 2001-style pragma declarations. The Quartus II software and the Encounter Conformal software support this style of pragma in different manners.

Stuck-at Registers

Quartus II integrated synthesis eliminates registers that have their output stuck at a constant value. Quartus II integrated synthesis gives a warning message and adds an entry to the corresponding report panel in the formal verification folder of the Analysis and Synthesis section of the Compilation Report. If Conformal LEC does not find the same optimizations, it can lead to unmapped points in the golden netlist. Example 17–3 illustrates the issue:

Example 17–3. Verilog HDL Example Showing Stuck at Registers

```
module stuck_at_example {clk, a,b,c,d,out};
input a,b,c,d,clk;
output out;
reg e,f,g;
    always @(posedge clk) begin
        e <= a and g;// e is stuck at 0
        g <= c and e;// g is stuck at 0
        f <= e | b;
    end
assign out = f and d;
endmodule
```

In this module description, registers `e` and `g` are tied to logic `0`. In this example, the Quartus II software generates the following warning message:

```
Warning: Reduced register "g" with stuck data_in port to stuck value GND
Warning: Reduced register "e" with stuck data_in port to stuck value GND
```

Quartus II integrated synthesis then adds a command to the formal verification scripts telling Conformal LEC that a register is stuck at a constant value, as shown in Example 17–4:

Example 17–4. Conformal LEC Script Showing Commands for Instance Equivalence

```
// report floating signals
// Instance-constraints commands for constant-value registers removed
// during compilation
// add instance constraints 0 e -golden
// add instance constraints 0 g -golden
```

The command is commented in the formal verification script, forcing the Encounter Conformal software to treat the register as stuck at a constant value, and potentially hiding a compilation error. You must verify that input to the `e` and `g` registers is constant in the design and uncomment the command to obtain accurate results.



Altera recommends recoding your design to eliminate "stuck-at" registers.

The stuck-at register information in this section also applies to the Synplify Pro flow.

ROM, LPM_DIVIDE, and Shift Register Inference

For the purpose of formal verification, the Quartus II integrated synthesis implements both ROM and shift registers in the form of LEs instead of using dedicated on-chip memory resources. Using LEs can be less area-efficient than inferring a megafunction that can be implemented in a RAM block. However, the Quartus II software generates a warning message indicating that the megafunction was not inferred. Quartus II integrated synthesis also reports a suggested ROM or shift register instantiation that enables you to either use the MegaWizard® Plug-In Manager to create the appropriate megafunction explicitly, or to isolate the corresponding logic in a separate entity that you can set as a black box. By setting black box properties on a particular module or entity, you are telling the formal verification tool not to peek inside the module or entity for formal verification. If the black box properties are set on the corresponding megafunction before synthesis, you can verify the megafunction with the Encounter Conformal software.

If the design contains division functionality, the Quartus II software infers an lpm_divide megafunction, which is treated as a black box for the purpose of formal verification.

RAM Inference

When the Quartus II software infers the LPM megafunction altsyncram from the RTL code, the Quartus II software generates the following warning message:

```
Created node "<mem_block_name>" as a RAM by generating altsyncram megafunction to implement
register logic with M512 or M4K memory block or M-RAM. Expect to get an error or a mismatch
for this block in the formal verification tool.
```

This warning is generated because the memory block (`altsyncram`) is a new instance in the post-fit netlist that is handled as a black box by the formal verification tool. However, no such instance exists in the original RTL design, resulting in mismatch or error reporting in the formal verification tool.

Latch Inference

A latch is implemented in the Quartus II integrated synthesis using a combinational feedback loop. The Encounter Conformal software infers a latch primitive in the Encounter Conformal library (DLAT) to implement a latch. This results in having a DLAT on the golden side and a combinational loop with a cut point on the revised side, leading to verification mismatches. The Quartus II software issues a warning message whenever a latch is inferred, and the Quartus II software adds an entry to the report panel in the Formal Verification folder of the Analysis and Synthesis report. Altera recommends that you avoid latches in your design; however, if latches are necessary, Altera recommends using the corresponding lpm_latch megafunction.



For more information about the problems related to latches, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Combinational Loops

If the design consists of an intended combinational loop, you must define an appropriate cut point for both the RTL and the post-fit Verilog Output netlist. A warning that a combinational loop exists in the design is found in the Formal Verification subfolder of the Quartus II software Analysis and Synthesis report.

For more information on issues with combinational loops, see "Known Issues and Limitations" on page 17–24.

Finite State Machine Coding Styles

When a state machine is inferred by the Encounter Conformal software, it uses sequential encoding as the default encoding when no user encoding is present. The Quartus II software selects the encoding most suited for the inferred state machine if the State Machine Processing Settings on the **Analysis and Synthesis Settings** page of the **Settings** dialog box is set to the default value Auto. Therefore, it is important to use the coding style described in the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook* on RTL when writing finite state Machines (FSMs). This allows the Quartus II integrated synthesis and the Encounter Conformal software to infer a similar state machine for the same RTL code.

Generating the Post-Fit Netlist Output File and the Encounter Conformal Setup Files

The following steps describe how to set up the Quartus II software environment to generate the post-fit Verilog Output netlist and the Encounter Conformal script for use in formal verification. With the exception of step 3, the steps are identical for both of the Synthesis tools:

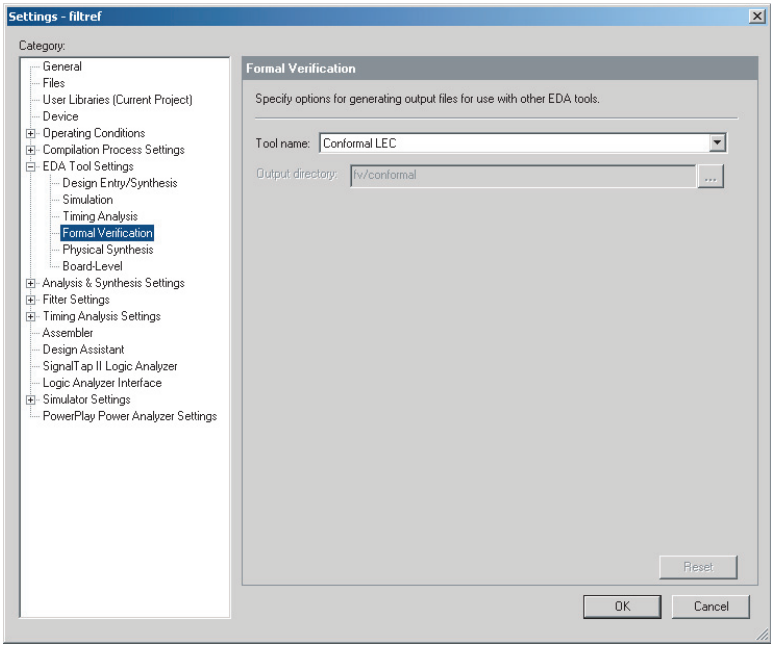
1. Create a new Quartus II project or open an existing project.
2. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
3. In the **Category** list, click **EDA Tool Settings**.

   If you are using the Quartus II integrated synthesis, perform the following steps:

   a. In the **Category** list, under **EDA Tool Settings**, select **Design Entry/Synthesis**. Select **<*None*>** from the Tool name list.

   b. In the **Category** list, under **EDA Tool Settings**, select **Formal Verification**. Select **Conformal LEC** from the Tool name list (Figure 17–3).

Figure 17–3. Compilation Process Settings



If you are using Synplify Pro, perform the following steps:

a. In the **Category** list, under **EDA Tool Settings**, select **Design Entry/Synthesis**. Select **Synplify Pro** from the Tool name list.

b. In the **Category** list, under **EDA Tool Settings**, select **Formal Verification**. Select **Conformal LEC** from the Tool name list.

4. In the **Category** list, select **Compilation Process Settings**. Under **Compilation Process Settings**, select **Incremental Compilation**.

   In the **Incremental Compilation** page, click **Full Incremental Compilation** to turn on Incremental Compilation.

   or

Turn on Incremental Compilation by typing the following Tcl command in the Quartus II software Tcl console:

Example 17–5. TcL Command to Turn On Full Incremental Compilation

```
set_global_assignment -name INCREMENTAL_COMPILATION \
FULL_INCREMENTAL_COMPILATION
```



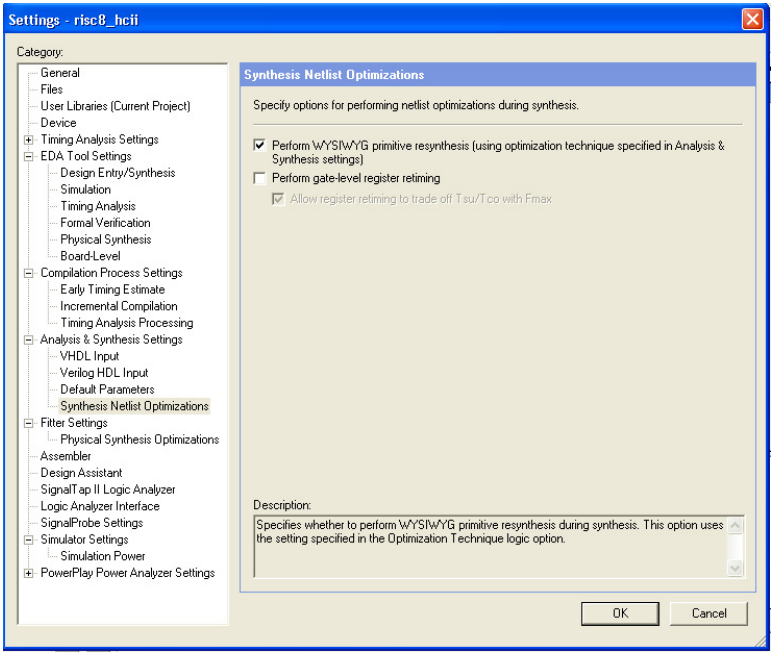
Altera requires that Incremental Compilation be turned **On** for Formal Verification, and that your design does not contain any user created partitions. Starting with Quartus II version 6.1 and later, the incremental compilation feature is **On** by default.

5. In the **Category** list, select **Analysis and Synthesis Settings** to expand the options list, and click **Synthesis Netlist Optimizations**. In the **Synthesis Netlist Optimizations** page, turn off **Perform gate-level register retiming** (Figure 17–4).



If **Perform gate-level register retiming** is not turned off, the Encounter Conformal script can display a different set of compare points, making the resulting netlist difficult to compare against the reference netlist file.

Figure 17–4. Synthesis Netlist Optimizations

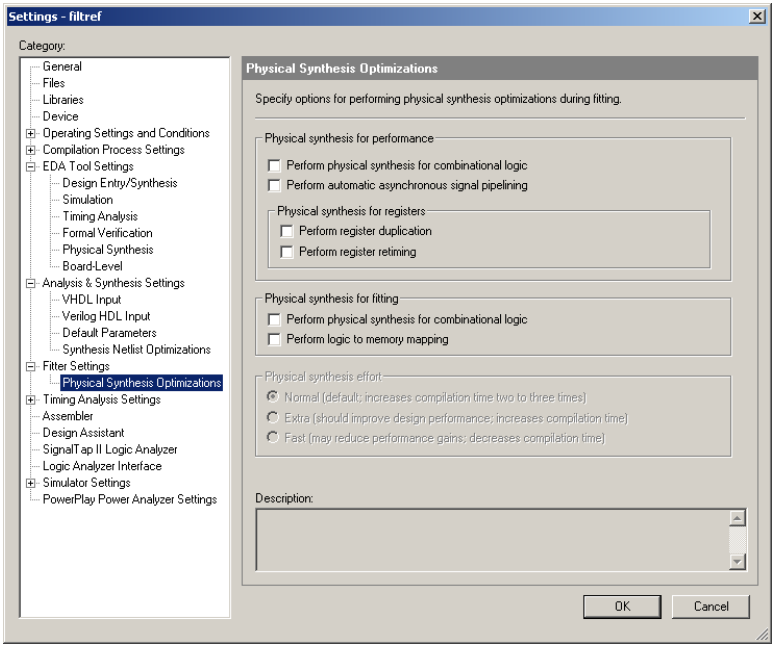


6. In the **Category** list, select **Fitter Settings,** and select **Physical Synthesis Optimizations**.

   a. Under **Physical synthesis for registers**, turn off **Perform register retiming**.

   b. Under Physical Synthesis for Fitting, turn off both **Perform physical synthesis for combinational logic** and **Perform logic to memory mapping** to prevent logic from being mapped to RAMs (Figure 17–5).

Figure 17–5. Fitter Settings



Retiming a design, either during the synthesis step or during the fitting step, usually results in moving and merging registers along the critical path and is not well supported by the equivalence checking tools. Because equivalence checkers compare the cone of logic terminating at registers, do not use retiming to move the registers during optimization in the Quartus II software.



If the options **Perform gate-level register retiming** (Figure 17–4) and **Perform register retiming** (Figure 17–5) are not turned off, the Encounter Conformal script can display a different set of compare points, making the resulting netlist difficult to compare against the reference netlist file. If you use retiming in your design during compilation, then you cannot generate a netlist for formal verification.



To learn more about physical synthesis, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

7. Perform a full compilation of the design. On the Processing menu, click **Start Compilation**, or click the **Start Compilation** icon on the Toolbar.

   If your golden netlist (VQM netlist from Synplify Pro or RTL) includes any design entity not having a corresponding formal verification model, that entity is handled as a black box whose boundary interface is preserved. There are three types of black boxes and required user actions, depending upon each circumstance. Table 17–3 describes these three types of black boxes and the required user actions in detail.

Table 17–3. Black Boxes and Required User Action

Type of Black Box	Required User Action
Altera library of parameterized modules (LPMs) and megafunctions (refer to Table 17–5 for a complete list).	No action required. The Quartus II software automatically black boxes the list of components and preserves the hierarchy.
Any parametrized entity other than those listed in Table 17–5.	User must black box the wrapper that instantiates the parameterized entity.
Non parameterized entities that the user wants to black box.	User can black box the entity itself.

You can also use Tcl commands or Quartus II GUI to set the black box property on the entities, which the formal verification tool does not compare.

Tcl Command

Use the following Tcl commands to preserve the boundary interface of a black box entity: `dram`.

Example 17–6. TcL Command to Create a Black Box

```
set_instance_assignment -name PRESERVE_HIERARCHICAL_BOUNDARY FIRM -to | -entity dram
set_instance_assignment -name EDA_FV_HIERARCHY BLACKBOX -to | -entity dram
```

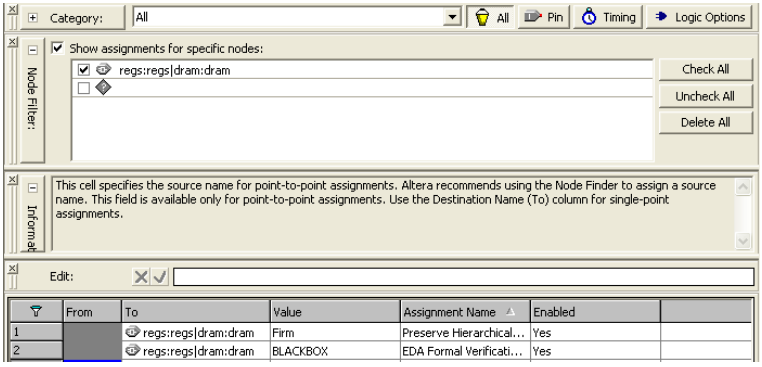
GUI

To preserve the boundary interface of an entity using the GUI, follow these steps:

1. Make an EDA Formal Verification Hierarchy assignment to the entity with the value `BLACKBOX`.

2. Make a Preserve Hierarchical Boundary assignment to the entity with the value `Firm` (Figure 17–6).

Figure 17–6. Setting the Black-Box Property on a Module



The Quartus II Software Generated Files, Formal Verification Scripts, and Directories

After successful compilation, the Quartus II software generates a list of files, formal verification scripts, and directories in the <*project_directory*>**fv/conformal/** directory (Table 17–4).

Table 17–4. The Quartus II Software Compiler-Generated Files and Directories (Part 1 of 2)

File or Directory	Name	Details
Verilog Output File	<*proj rev*>**.vo**	The Quartus II software-generated netlist for formal verification.

Table 17–4. The Quartus II Software Compiler-Generated Files and Directories (Part 2 of 2)

File or Directory	Name	Details
Script file	<*proj rev*>**.ctc**	The <*proj rev*>**.ctc** file references <*proj rev*>**.clg** and <*proj rev*>**.clr** that read the library files and black box descriptions. The <*proj rev*>**.ctc** file also references the <*proj rev*>**.cmc** file containing information about the mapped points. *(1)*
	<*proj rev*>.**cec**	The <**proj rev**>.**cec** file contains the information for instance equivalences.
	<*proj rev*>.**cep**	The <**proj rev**>.**cep** file contains the information for black box pin equivalences in the design.
	<*proj rev*>.**cmp**	The <*proj rev*>.**cmp** file contains the information for the black box pin mapping between the golden and revised sides. *(2)*
	<*proj rev*>**.cmc**	The <*proj rev*>**.cmc** file contains information about the additional points to be mapped in addition to the points selected by the tool.
	<*proj rev*>_trivial**.cmc**	This <*proj rev*>_trivial**.cmc** file contains mapping information for all the key points in the design. *(3)*
	<*proj rev*>**.clr**	The <*proj rev*>**.clr** file contains information about the macros and libraries for the revised design.
	<*proj rev*>**.clg**	The <*proj rev*>**.clg** file contains information about the macros and libraries for the golden design.
blackboxes directory	**<***project directory***>/fv/conformal/<***project rev***>_blackboxes**	This directory contains top-level module descriptions for all the user-defined black box entities and contains modules with definitions other than Verilog or VHDL, for example, Block Design File (**.bdf**) in the design directory **<***project directory***>/fv/conformal/<***project rev***>_blackboxes**

Notes to Table 17–4:

(1) This file is used with the Encounter Conformal software.

(2) This file is called from the <*proj rev*>.**ctc** script file. By default, the line where this file is called is commented out. These files are only useful for HardCopy II device families.

(3) In some cases, Encounter Conformal software performs incorrect key point mapping, resulting in formal verification mismatches. To overcome the verification mismatches, the Quartus II software writes out the <*proj rev*>**_trivial.cmc** file that contains mapping information for all the key points in the design. Reading this file during the formal verification setup can result in increased run time. Therefore, the Quartus II software writes out the top-level script file <*proj rev*>**.ctc** with the command to read the <*proj rev*>**_trivial.cmc** file commented. If the formal verification results are not acceptable, the user can uncomment the command and read the <*proj rev*>**_trivial.cmc** file. The command in the <*proj rev*>**.ctc** file is:

```
//Trivial mappings with same name registers
//read mapped points $PROJECT/fv/conformal/<proj rev>_trivial.cmc
```

The script file contains the setup and constraints information to be used with the formal verification tool. The file <*entity*>**.v** in the **blackboxes** directory contains the module description of entities that are not defined in the formal verification library. The file also contains entities that you

specify as black boxes. For example, if there is a reference to a black box for an instance of the altdpram megafunction in the design, the **blackboxes** directory does not contain a module description for the altdpram megafunction because it is defined in the **altdpram.v** file of the formal verification library. When a module does not have an RTL description, or the description exists only in the formal verification library and you do not want to compare the module using formal verification, a file containing only the top-level module description with port declaration is written out to the **blackboxes** directory and read into the Encounter Conformal software.

Understanding the Formal Verification Scripts for Encounter Conformal

The Quartus II software generates scripts to use with the Encounter Conformal Logic Equivalence Check (LEC) software. This section elaborates on the details of the Encounter Conformal commands used within the scripts to help you compare the revised netlist with the golden netlist. In most cases, you do not need to add any more Encounter Conformal constraints to verify your netlists. Also, a sample script generated by the Quartus II software is provided at the end of the chapter.

The Encounter Conformal Commands within the Quartus II Software-Generated Scripts

The value for the variable `QUARTUS` is the path to the Quartus II software installation directory:

```
setenv QUARTUS <Quartus Installation Directory>
```

The Quartus II software assigns the current working directory of the project to the `PROJECT` variable. Use this variable to change the project directory to the directory where the design files are installed when moving from a UNIX to a Windows environment, or vice versa:

```
setenv PROJECT <Quartus Project Directory>
```

The following command reads both the golden and the revised netlists, along with the appropriate library models:

```
read design <design files>
```



You must update the project location when the files are moved from the Windows environment to the UNIX environment.

The post place-and-route netlist from the Quartus II software might contain net and instance names that are slightly different from those of the golden netlist. By using the following command, the Quartus II

software defines temporary substitute string patterns enabling the Encounter Conformal software to automatically map key points when the names are not the same:

```
add renaming rule <rule>
```

The Encounter Conformal LEC software employs three name-based methods to map key points to compare the revised netlist with the golden netlist. Scripts set the correct method to get the best results.

```
set mapping method <mapping_rule>
```

The Quartus II software performs several optimizations, including optimizing the registers whose input is driven by a constant. Under these circumstances, for the formal verification software to compare the netlists properly, the command `set flatten model` is used with the option `seq_constant`.

```
set flatten model <flattening_rule>
```

When you use the command `report black box`, verify that the following modules are listed as black boxes, along with any of the modules black boxed by the user, in both the golden and revised netlists:

- LPMs and megafunctions without the formal verification models
- Encrypted IP functions
- Entities not implemented in Verilog HDL or VHDL

Use the following command to set the same implementation on multipliers for both the golden and revised netlists:

```
set multiplier implementation <implementation_name>
```

If there are any combinational loops or instances of `LPM_LATCH`, the Quartus II software cuts the loop at the same point using the following command on both the golden and revised netlists:

```
add cut point
```

The Encounter Conformal software does not always automatically map all the keypoints, or can incorrectly map some keypoints. To help the Encounter Conformal software successfully complete the mapping process, the Quartus II software records optimizations performed on the netlist as a series of `add mapped points` in the Encounter Conformal <*file_name*>**.cmc** script.

```
add mapped points <key_points>
```

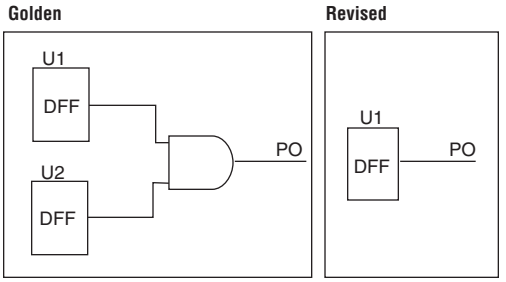
There are situations where the inverter in front of the register gets moved after the register. In this situation, the following command is used:

```
add mapped points <key_points> -invert
```

The following command reads in the mapped point information from the specified file:

```
read mapped points <file_name>.cmc
```

Figure 17–7. Instance Equivalence



During the process of optimization, the Quartus II software might merge two registers into one (Figure 17–7). The Quartus II software informs the formal verification tool that the U1 and U2 registers are equivalent to each other using the following command:

```
add instance equivalence <instance_pathname ..> [-Golden]
```

If the register duplication takes place, the following command is used:

```
add instance equivalence <instance_pathname ..>
[-revised]
```

The following command is used when the inverter is moved beyond the register along with either register duplication or merging:

```
add instance equivalences <instance_pathname>
[-invert <instance_pathname>]
```

At times, the register output is driven to a constant, either `logic 0` or `logic 1`. The Quartus II software sets the value of the register to a constraint using the `add instance constraint` command. For more information about this command, refer to "Stuck-at Registers" on page 17–7.

```
add instance constraint <constraint_value>
```

Comparing Designs Using Encounter Conformal

This section addresses using the Encounter Conformal software to compare designs; that is, how to prove logical equivalence between two versions of the design.

Black Boxes in the Encounter Conformal Flow

The Quartus II software usually generates a flattened netlist. However, there are some modules in the design that must be treated differently. The following is a list of some of these modules:

- LPMs and megafunctions without formal verification models
- Encrypted IP functions
- Entities not implemented in Verilog HDL or VHDL

To perform equivalence checking of a design between its version consisting of the modules listed above and its implemented version, the modules have to be treated as black boxes by the Encounter Conformal software. To facilitate the formal verification flow, the Quartus II software reconstructs the hierarchy on the black boxes with a port interface that is identical to the module on the golden side of the design.

Verilog Output netlist files written by the Quartus II software also contain the black box hierarchy when you make the following assignments for a module:

- An EDA Formal Verification Hierarchy assignment with the value `BLACKBOX`
- A Preserve Hierarchical Boundary assignment with the value `Firm` (Figure 17–6)

If these two assignments are not made for a module, the Quartus II software implements that module with logic cells. When this happens, the Verilog Output netlist file no longer contains the black box hierarchy and does not preserve the port interface, resulting in a mismatch within the Encounter Conformal software.

Running the Encounter Conformal Software

To run the Encounter Conformal software, use its GUI or a system command prompt, and use the CTC script generated by the Quartus II software.

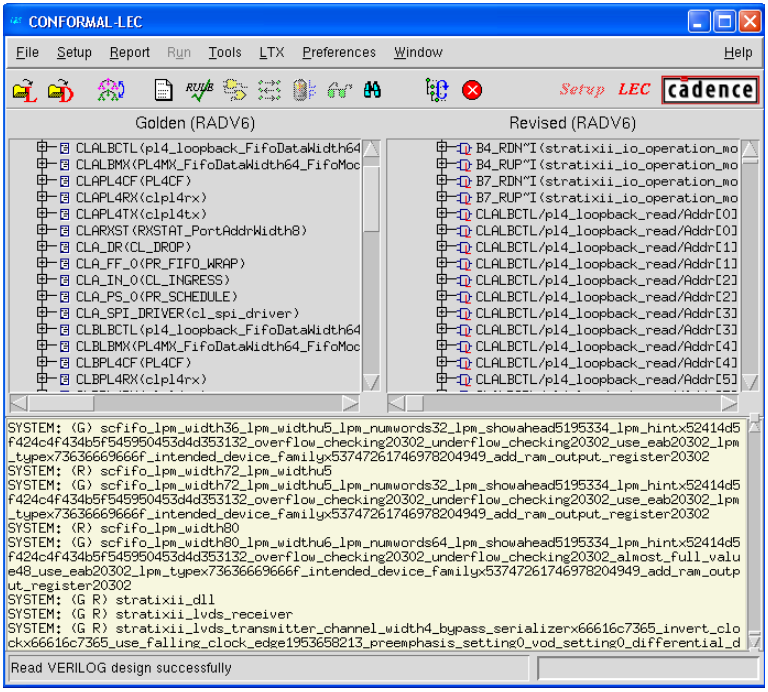
Running the Encounter Conformal Software from the GUI

To run the Encounter Conformal software from the GUI, follow these steps:

1. Open the Encounter Conformal software.
2. On the File menu, click **Do Dofile**.
3. Select the file **<***path to project directory***>/fv/conformal/**<*proj rev*>**.ctc**.

The Encounter Conformal software GUI displays the comparison results (Figure 17–8). The Golden window displays the original RTL description or the post synthesis VQM netlist from Synplify Pro, and the Revised window displays the information of the post-fit netlist generated by the Quartus II software. The message section at the bottom of the window reports the verification results and the number of unmapped and non-equivalent points found in the design.

Figure 17–8. Encounter Conformal Software GUI Display of Functional Comparisons



To investigate the verification results, click the **Mapping Manager** icon in the toolbar, or on the Tools menu, click **Mapping Manager**. The Encounter Conformal software reports the mapped, unmapped, and compared points in the Mapped Points, Unmapped Points, and Compared Points windows, respectively.

For more information about how to diagnose non-equivalent points, refer to the Encounter Conformal software user documentation.

Running the Encounter Conformal Software From a System Command Prompt

To run the Encounter Conformal Software without using the GUI, type the command shown in Example 17–7 at a system command prompt.

Example 17–7. Conformal LEC Command to Run Formal Verification

```
lec -dofile /<path to project directory>/fv/conformal/<proj rev>.ctc -nogui ↵
```

To get a downloadable design example showing the formal verification flow with Quartus II software, go to www.altera.com/support/examples/quartus/exm-formal-verification.html.



To learn more about the latest debugging tips and solutions for formal verification flow between Cadence Conformal LEC tool and Quartus II software, go to www.altera.com and perform an advanced search with keywords "formal verification."

Known Issues and Limitations

The following known issues and limitations can occur when using the formal verification flow described in this chapter:

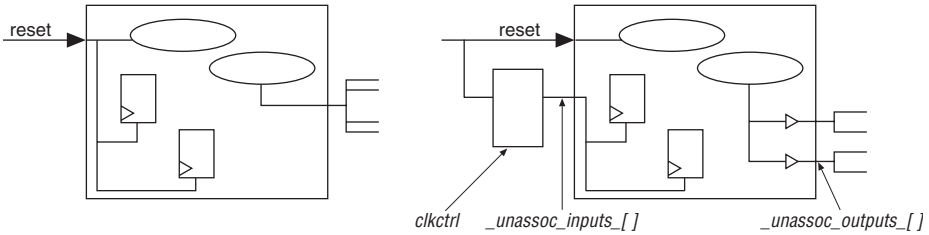
- When a port on a black box entity drives two or more signals within the black box, the Quartus II software pushes the connections outside of the black box, and creates that many ports on the black box. This problem is only associated with Stratix II and HardCopy II designs.

  The additional ports on the black box are named `_unassoc_inputs_[]` and `_unassoc_outputs_[]` (Figure 17–9). This issue is generally associated with reset and enable signals. Figure 17–9 shows an example in which the reset pin is split into two ports outside of the black box and the `_unassoc_inputs_[]` is driven by the clkctrl block. In such situations, the Verilog Output netlist generated by the Quartus II software has signals driving these black box ports, but golden RTL does not contain any signals to drive the `_unassoc_inputs_[]`, resulting in a formal verification mismatch of the black box. The black box module definition generated by the Quartus II software in the directory <*Quartus_project*>\fv\conformal*_blackboxes contains these additional `_unassoc_inputs_[]` and `_unassoc_outputs_[]` ports. This black box module is read on

both the golden and revised sides of the design, which results in unconnected ports on the golden side and formal verification mismatches.

Figure 17–9 shows the creation of `_unassoc_inputs_[]` and `_unassoc_outputs_[]` for the reset signal.

Figure 17–9. Creation of _unassoc_inputs_[] and _unassoc_outputs_[]

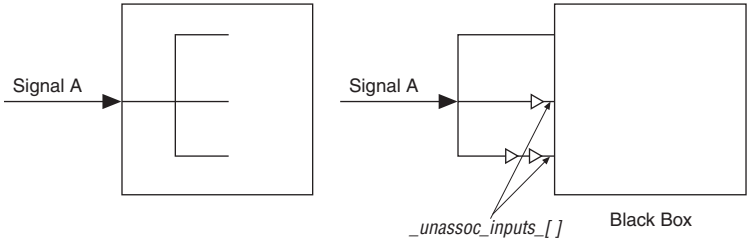


Another common occurrence of this issue is in HardCopy II designs. Whenever a port drives large fan-out within the black box, the Quartus II software inserts a buffer on the net and moves the logic outside of the black box (Figure 17–10).

To fix the problem of `unassoc_input_[]` ports causing blackbox mismatches, use Cadence Conformal commands to change the type of the blackbox `unassoc_input_[]` keypoint to a primary output keypoint, and then marking the appropriate pin equivalences. Similarly, to fix the problem of register mismatches due to `unassoc_output_[]` pins from blackboxes, use Conformal commands to change the type of the blackbox `unassoc_output_[]` keypoint to a primary input, and then marking equivalent pins as such. The commands to perform these actions are written in the <*proj rev*>**.cep** file.

Figure 17–10 shows the creation of _unassoc_inputs_[] for a signal with large fan-out.

Figure 17–10. Creation of _unassoc_inputs_[] for a Signal with Large Fan-out



- In designs with combinational feedback loops, the Encounter Conformal software can insert extra cut points in the revised netlist, causing unmapped points and ultimately verification mismatches.
- For Cyclone II designs, Conformal LEC may report non-equivalent flipflops and extra cut points for the revised (post-fit) design when your HDL source code instantiates the lpm_ff primitive with an asynchronous load signal aload (with or without any other asynchronous control signals) and when the asynchronous clear signal aclr and asynchronous set signal aset are used together. To avoid this problem, ensure that there is a wrapper module or entity around the lpm_ff instantiation, and black box the module or entity that instantiates the lpm_ff primitive.
- For Stratix III designs, the Cadence Conformal LEC software creates cut points for the combinational loops on the golden side and may fail equivalence checking due to improper mapping. The combinational loops are due to logic around the registers emulating multiple set, resets, or both. These cut points are also reported during the mapping step in Quartus II software with Warning messages. You can manually add Cadence Conformal commands to add cut points, which result in proper mapping and formal verification.
- To perform formal verification, certain synthesis optimization options, such as register retiming, optimization through black box hierarchy boundaries, and disabling the ROM and shift register inference, are turned off, which can have an impact on the area resource and performance.
- RAM and ROM instantiations, inferences, or both are not verified using formal verification.

- Incremental Compilation for the purpose of formal verification does not support user-created design partitions.
- Formal verification does not support clear box netlist due to unconnected ports on its WYSIWYG instances.
- Formal verification does not support VHDL megafunction variations due to undriven ports on the megafunctions.
- When a black box contains bidirectional ports, the Quartus II software fails to reconstruct the hierarchy. For this reason, the black box is represented by a flat netlist, resulting in formal verification mismatches.
- ROMs in the design have to be black boxed before compilation using Quartus Integrated Synthesis, because the Quartus II software may perform some optimizations on the ROM, resulting in Formal Verification mismatches.
- Conformal may report mismatches or abort comparison of some key points when a DSP megafunction is implemented in LEs by the Quartus II software due to implicit optimizations within the DSP and the complexity of the multiplier logic in terms of LEs.
- Unused logic optimized within and around a black box by the Quartus II software can result in a black-box interface different from the interface in the synthesized VQM netlist.

Conclusion

Formal verification software enables verification of the design during all stages from RTL to placement and routing. Verifying designs takes more time as designs increase in size. Formal verification is a technique that helps reduce the time needed for your design verification cycle.

Black Box Models

The black box models are interface definitions of entities, such as primitives, atoms, LPMs, and megafunctions. These models have a parameterized interface, and do not contain any definition of behavior. They are specifically designed and tested to work with the Encounter Conformal software, which uses these models along with your design to generate black boxes for instances of the entity with varying sets of parameters in the design. Table 17–5 describes the supported black box models. Besides these black box models, you can set a black box property on a specific module or entity as explained earlier in this chapter.

Table 17–5. Supported Black Box Models (Part 1 of 3)

Entity Type	Entity Names
Megafunctions	alt3pram, altaccumulate, altfp_mult, altsqrt, altlvds_rx, altlvds_tx, altshift_taps, sld_virtual_jtag sld_virtual_jtag_basic dcfifo, scfifo, altsyncram, altsqrt
LPMs	lpm_add_sub, lpm_divide

Table 17–5. Supported Black Box Models (Part 2 of 3)

Entity Type	Entity Names
Atoms *(1)*	Cyclone: cyclone_crcblock, cyclone_jtag, cyclone_pll, cyclone_ram_block, cyclone_asmiblock, cyclone_dll
	Stratix: stratix_crcblock, stratix_jtag, stratix_lvds_receiver, stratix_lvds_transmitter, stratix_pll, stratix_rublock, stratix_ram_block, stratix_dll
	Stratix II: stratixii_crcblock, stratixii_jtag, stratixii_lvds_receiver, stratixii_lvds_transmitter, stratixii_pll, stratixii_rublock, stratixii_ram_block, stratixii_asm_block, stratixii_dll, stratixii_termination, stratixii_asmiblock
	Stratix GX: stratixgx_crcblock, stratixgx_jtag, stratixgx_lvds_receiver, stratixgx_lvds_transmitter, stratixgx_pll, stratixgx_rublock, stratixgx_ram_block, stratixgx_dll
	Stratix II GX: stratixiigx_hssi_receiver, stratixiigx_hssi_transmitter, stratixiigx_hssi_central_management_unit, stratixiigx_hssi_cmu_pll, stratixiigx_hssi_cmu_clock_divider, stratixiigx_hssi_refclk_divider, stratixiigx_hssi_calibration_block, stratixiigx_crcblock, stratixiigx_ram_block, stratixiigx_lvds_transmitter, stratixiigx_lvds_receiver, stratixiigx_pll, stratixiigx_dll, stratixiigx_jtag, stratixiigx_asmiblock, stratixiigx_termination, stratixiigx_rublock
	Cyclone II: cycloneii_asmiblock, cycloneii_clk_delay_ctrl, cycloneii_clkctrl, cycloneii_jtag, cycloneii_pll, cycloneii_ram_block
	Arria GX: arriagx_asmiblock, arriagx_crcblock, arriagx_dll, arriagx_hssi_calibration_block, arriagx_hssi_central_management_unit, arriagx_hssi_cmu_clock_divider, arriagx_hssi_cmu_pll, arriagx_hssi_receiver, arriagx_hssi_refclk_divider, arriagx_hssi_transmitter, arriagx_jtag, arriagx_lvds_receiver, arriagx_lvds_transmitter, arriagx_pll, arriagx_ram_block, arriagx_rublock, arriagx_termination
	HardCopy II: hardcopyii_crcblock, hardcopyii_dll, hardcopyii_jtag, hardcopyii_lvds_receiver, hardcopyii_lvds_transmitter, hardcopyii_pll, hardcopyii_ram_block, hardcopyii_termination

Table 17–5. Supported Black Box Models (Part 3 of 3)

Entity Type	Entity Names
	Stratix III: stratixiii_asmiblock, stratixiii_crcblock, stratixiii_jtag, stratixiii_lvds_receiver, stratixiii_lvds_transmitter, stratixiii_mlab_cell, stratixiii_pll, stratixiii_ram_block, stratixiii_rublock, stratixiii_termination, stratixiii_tsdblock

Note to Table 17–5:
(1) The entity names are given for the specific device family listed.

Conformal Dofile/Script Example

The following example script (17–8), generated by the Quartus II software, lists some of the setup commands used in Conformal LEC software:

Example 17–8. Conformal LEC Script

```
// Copyright (C) 1991-2007 Altera Corporation
// Your use of Altera Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logi
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Altera Program License
// Subscription Agreement, Altera MegaCore Function License
// Agreement, or other applicable license agreement, including,
// without limitation, that your use is for the sole purpose of
// programming logic devices manufactured by Altera and sold by
// Altera or its authorized distributors.  Please refer to the
// applicable agreement for further details.

// Script generated by Quartus II

reset
set system mode setup
set log file mfs_3prm_1a.fv.log -replace
set naming rule "%s" -register -golden
set naming rule "%s" -register -revised
// Naming rules for Verilog
set naming rule "%L.%s" "%L[%d].%s" "%s" -instance
set naming rule "%L.%s" "%L[%d].%s" "%s" -variable
// Naming rules for VHDL
// set naming rule "%L:%s" "%L:%d:%s" "%s" -instance
// set naming rule "%L:%s" "%L:%d:%s" "%s" -variable
// set undefined cell black_box -both
// These are the directives that are not supported by the QIS RTL to gates FV flow
set directive off verplex ambit
set directive off assertion_library black_box clock_hold compile_off compile_on
set directive off dc_script_begin dc_script_end divider enum infer_latch
set directive off mem_rowselect multi_port multiplier operand state_vector template
add notranslate module alt3pram -golden
```

```
add notranslate module alt3pram -revised
setenv QUARTUS /data/quark/build/ajaishan/quartus
setenv PROJECT
/net/quark/build/ajaishan/quartus_regtest/eda/fv/conformal/synplify/stratix/mfs_3prm_1a_v1
_/mfs_3prm_1a/qu_allopt
read design \
    $QUARTUS/eda/fv_lib/vhdl/dummy.vhd \
    -map lpm $QUARTUS/eda/fv_lib/vhdl/lpms \
    -map altera_mf $QUARTUS/eda/fv_lib/vhdl/mfs \
    -map stratix $QUARTUS/eda/fv_lib/vhdl/stratix \
    -vhdl -noelaborate -golden
read design \
    -file $PROJECT/fv/conformal/mfs_3prm_1a.clg \
    $PROJECT/p3rm_block.v \
    $PROJECT/mfs_3prm_1a.v \
    -verilog2k -merge none -golden
read design \
    $QUARTUS/eda/fv_lib/vhdl/dummy.vhd \
    -map lpm $QUARTUS/eda/fv_lib/vhdl/lpms \
    -map altera_mf $QUARTUS/eda/fv_lib/vhdl/mfs \
    -map stratix $QUARTUS/eda/fv_lib/vhdl/stratix \
    -vhdl -noelaborate -revised
read design \
    -file $PROJECT/fv/conformal/mfs_3prm_1a.clr \
    $PROJECT/fv/conformal/mfs_3prm_1a.vo \
    -verilog2k -merge none -revised
// add ignored inputs _unassoc_inputs_* -all -revised
add renaming rule r1 "~I\/" "\/" -revised
add renaming rule r2 "_I\/" "\/" -revised
set multiplier implementation rca -golden
set multiplier implementation rca -revised
set mapping method -name first
set mapping method -nounreach
set mapping method -noreport_unreach
set mapping method -nobbox_name_match
set flatten model -seq_constant
set flatten model -nodff_to_dlat_zero
set flatten model -nodff_to_dlat_feedback
set flatten model -nooutput_z
set root module mfs_3prm_1a -golden
set root module mfs_3prm_1a -revised
report messages
report black box
report design data
// report floating signals
dofile $PROJECT/fv/conformal/mfs_3prm_1a.cec
// dofile $PROJECT/fv/conformal/mfs_3prm_1a.cep
// Instance-constraints commands for constant-value registers removed
// during compilation
set system mode lec -nomap
read mapped points $PROJECT/fv/conformal/mfs_3prm_1a.cmc
// Trivial mappings with same name registers
// read mapped points $PROJECT/fv/conformal/mfs_3prm_1a_trivial.cmc
// dofile $PROJECT/fv/conformal/mfs_3prm_1a.cmp
map key points
remodel -seq_constant -repeat
add compare points -all
compare
usage
// exit -f
```

Referenced Documents

This chapter references the following documents:

- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 17–6 shows the revision history for this chapter.

Table 17–6. Document Revision History

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Updated Introduction section on page 17–1. • Updated Known Issues and Limitations section on page 17–24. • Updated Table 17–1. • Updated Table 17–5. • Updated Figure 17–10.	Updated for Quartus II software version 7.2.
May 2007 v7.1.0	• Updated Formal Verification Design Flow section on page 17–2. • Updated Generating the Post-Fit Netlist Output File and Encounter Conformal Setup Files section on page 17–10. • Updated Understanding the Formal Verification Scripts for Encounter Conformal section title on page 17–18. • Updated Known Issues and Limitations on page 17–24. • Renamed Tcl Sample Script to Conformal Dofile/Script Example and updated section on page 17–29. • Added Referenced Documents on page 17–31. • Removed Debugging Tips section. • Updated Figure 17–3. • Updated Figure 17–5. • Updated Table 17–1. • Updated Table 17–4. • Updated Table 17–5.	Updated for Quartus II software version 7.1.
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Changed date only.	Updated for Quartus II software version 6.1
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0.	—
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 15 was previously Chapter 13 in version 5.0.	—
May 2005 v5.0.0	New functionality for Quartus II software 5.0.	—
January 2005 v1.0	Initial release.	—

18. Synopsys Formality Support

QII53015-7.2.0

Introduction

Formal verification of FPGA designs is gaining momentum as multi-million System-on-a-Chip (SoC) designs are targeted at FPGAs. Use the Formality software to easily verify logic equivalency between the RTL and DC FPGA post-synthesis netlist, and between the DC FPGA post-synthesis netlist and Quartus II post-place-and-route netlist. Beginning with version 4.2, the Quartus® II software interfaces with EDA tools including the Formality and DC FPGA software from Synopsys.

This chapter discusses the following:

- "Formal Verification"
- "Formal Verification Support" on page 18–2
- "Generating Post-Synthesis Netlist for Formal Verification" on page 18–3
- "Generating the VO File and Formality Script" on page 18–4
- "Quartus II Scripts for Formality" on page 18–11
- "Comparing Designs Using the Formality Software" on page 18–11
- "Known Issues and Limitations" on page 18–12

Formal Verification

Formal verification uses exhaustive mathematical techniques to verify design functionality. There are two types of formal verification: equivalence checking and model checking. This section discusses equivalence checking.

Equivalence Checking

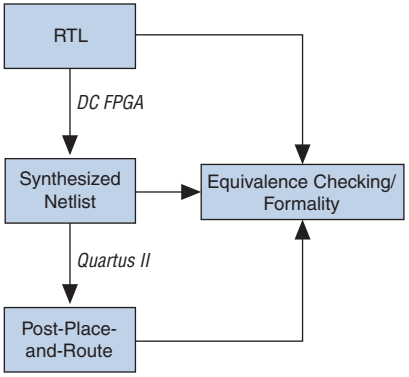
Equivalence checking compares the logical equivalence between the original design and the modified or revised design using mathematical techniques. This method reduces the verification time several-fold compared to the traditional method of performing verification using test vectors. Using a formal verification methodology provides the following key advantages:

- Faster time-to-market
- No testbenches or test vectors
- Results in hours compared to days using traditional verification methods

Formal Verification Support

The Quartus II software supports formal verification using the Formality software for the DC FPGA Synthesis tool as shown in Figure 18–1.

Figure 18–1. Equivalence Checking in the FPGA Design Flow



EDA Tools and Device Support

The formal verification flow using the Quartus II software and Synopsys Formality software requires the following software versions:

- Quartus II software, beginning with version 4.2
- Synopsys DC FPGA software, beginning with version W2005.03_EA1
- Synopsys Formality software, beginning with version 2004.12

The formal verification flow, using the Quartus II and Synopsys Formality software, supports Solaris and Linux platforms, and supports Stratix series devices.

Formal Verification Between RTL and Post-Synthesis Netlist

The first step in the FPGA design flow is to synthesize the RTL code using the DC FPGA to generate the synthesized verilog netlist. Equivalence checking using formal verification is performed between the RTL and the synthesized netlist to make sure the synthesis tool has not altered the original functionality of the design.



For more information on how to use the DC FPGA software for synthesizing Altera device designs, refer to the *Synopsys Design Compiler FPGA Support* chapter in volume 1 of the *Quartus II Handbook*.

Generating Post-Synthesis Netlist for Formal Verification

During the synthesis process, the DC FPGA synthesis tool performs operations such as:

- Modifying the net/instance names
- Register duplication
- State machine extraction by different methods

Changes caused by these synthesis operations cause comparison point matching issues and false verification failures. In order to make sure that the Formality software is aware of the design transformations performed during the synthesis, the DC FPGA software writes out a Synopsys setup verification file (**.svf**) to be read into the Formality software. To ensure the SVF constraint file contains all the formal verification setup constraints, you need to set certain commands in the DC FPGA software before compiling the design as detailed in the following section.

DC FPGA Software Settings

The Formality software does not support the **register merging** or **register retiming** synthesis operations, which are off by default, but it is necessary to verify that these settings are turned off during synthesis. Some of the commands necessary to turn off these options and generate a valid Verilog netlist for the formal verification purpose are described in this section.



For more information on creating the Tcl script file to perform synthesis, refer to the *DC FPGA User Guide* or the *Synopsys Design Compiler FPGA Support* chapter in volume 1 of the *Quartus II Handbook*.

To set most of the required synthesis settings to generate a valid formal verification netlist, use the following command:

```
set_fpga_defaults -formality <architecture_name>
```

For example:

```
set_fpga_defaults -formality altera_stratix
```

To view all of the settings performed by this command, add `-verbose` to this command. In addition, you will need to execute the additional commands shown in Table 18–1.

Table 18–1. Commands and Affect of Each Command

Command	Affect
`set verilogout_write_constant_nets true`	Add this command at the beginning of the script to allow unconnected nets to be driven by either power or ground.
`change_names -rule verilog -hierarchy`	This command must be added after the compile command to set the Verilog naming rule to the output netlist for all levels of hierarchy.
`set_verification_friendly_mode -filename \ <top_level>.svf -append -allow_override`	This command helps DC_FPGA to write out a SVF constraint file to be read into the Formality software.
`write -hier -f verilog -o $outputdir/<top_level>.v`	This command writes out a Verilog netlist for Formal Verification.

For a sample DC FPGA script that is ready for compilation, refer to "Tcl Sample Script" on page 18–13.

Post synthesis Verilog netlist for formal verification can be generated by executing the Tcl script either in `fpga_vision` (GUI) or `fpga_shell-t` (command line).



For comparing RTL against post-synthesis netlist using the Formality software, refer to the *DC FPGA Software User Guide*.

Generating the VO File and Formality Script

The following steps describe how to set up the Quartus II software environment to generate the place-and-route, post-place-and-route VO netlist file, and Formality script compatible for formal verification.

1. Create a new Quartus II project or open an existing project.
2. On the Assignments menu, click **Settings**. The **Settings** dialog box is shown.
3. In the Category list, select **Files**. The **Files** page is shown.
4. Highlight the input file by clicking on it, then click **Properties** and select **Verilog Quartus Mapping File**. Click **OK**.

5. In the **Category** list, select **Design entry/synthesis** under **EDA Tool Settings**.

6. In the **Tool name** list, select **Design Compiler FPGA** (Figure 18–2).

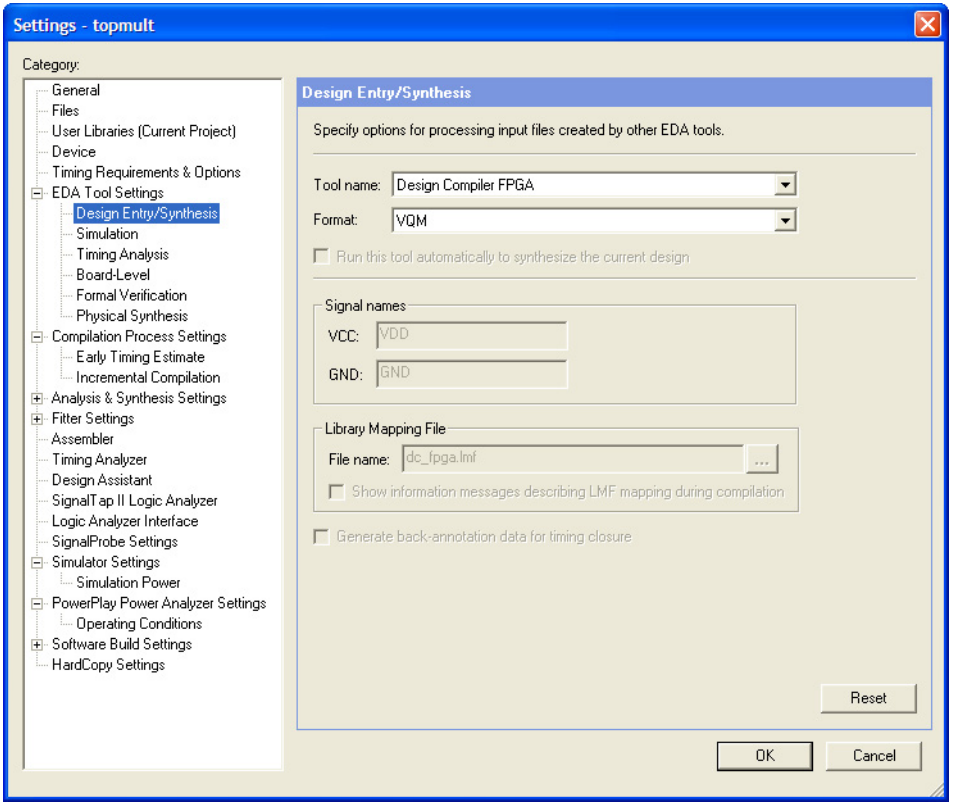
These settings can also be performed using the following Tcl commands:

```
set_global_assignment -name VQM_FILE
<verilog_file_from_dc_fpga>

set_global_assignment -name \
EDA_DESIGN_ENTRY_SYNTHESIS_TOOL "Design Compiler FPGA"

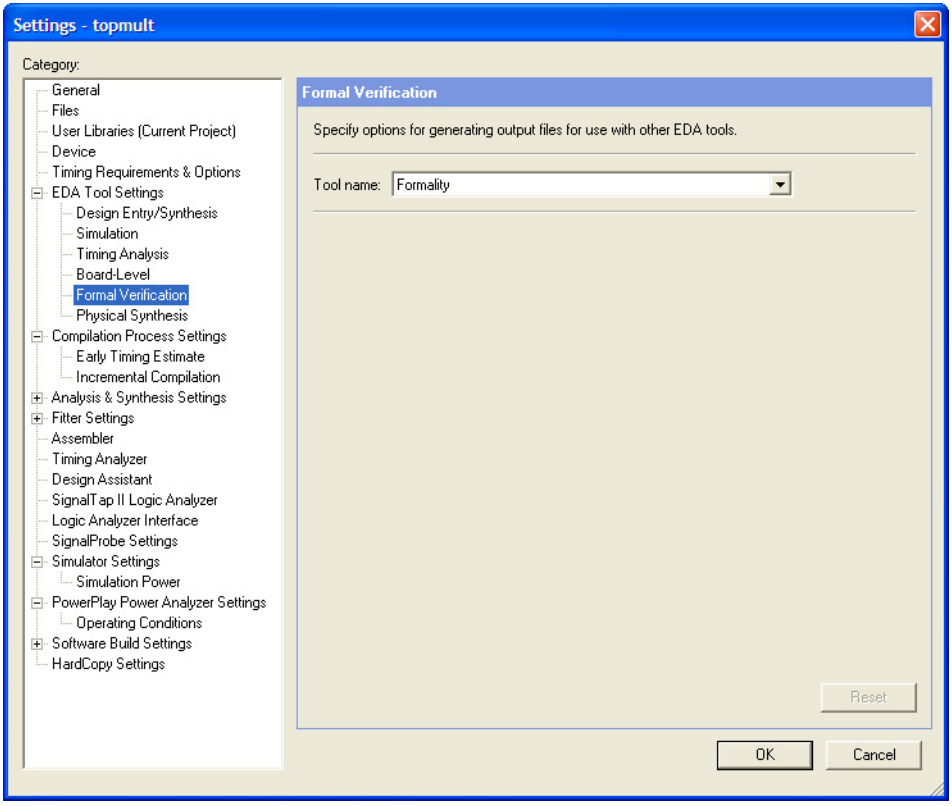
set_global_assignment -name EDA_LMF_FILE \
dc_fpga.lmf -section_id eda_design_synthesis
```

Figure 18–2. EDA Tools Selection



7. In the **Category** list, select **Formal verification**. In the **Tool name** list, select **Formality** (Figure 18–3).

Figure 18–3. EDA Tools Selection



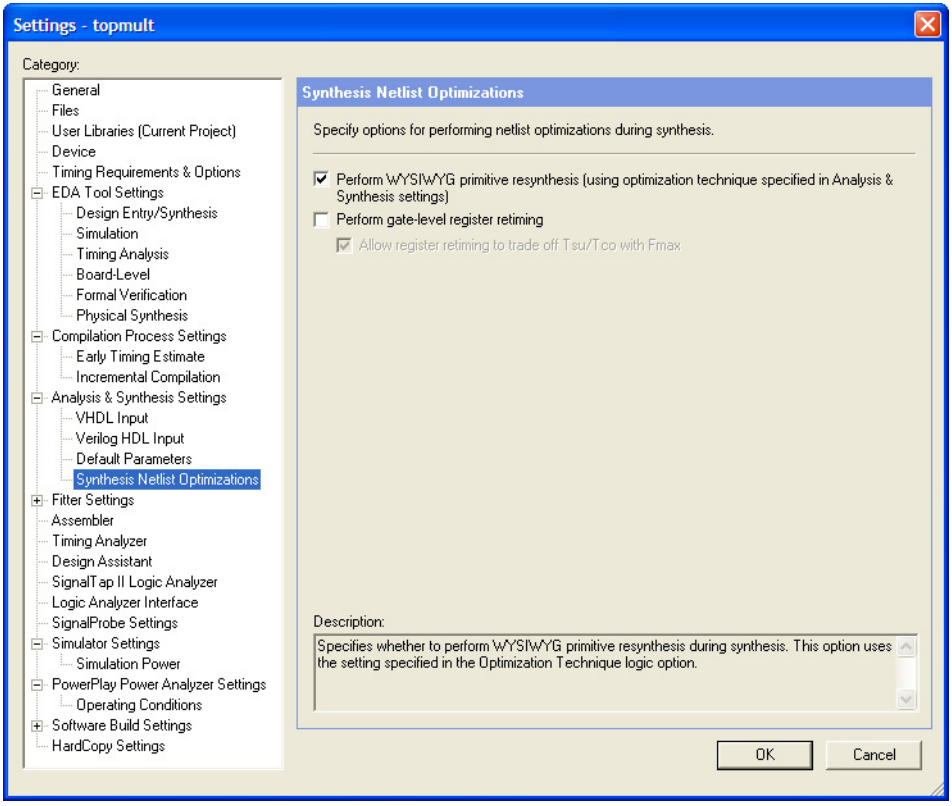
8. Click **OK**.

9. From the Assignments menu, click **Settings**. The **Settings** dialog box is shown.

10. In the **Category** list, click the **+** icon to expand **Analysis and Synthesis Settings** and select **Synthesis Netlist Optimizations**. The **Synthesis Netlist Optimizations** page is shown.

11. Turn off the **Perform gate-level register retiming** option (Figure 18–4).

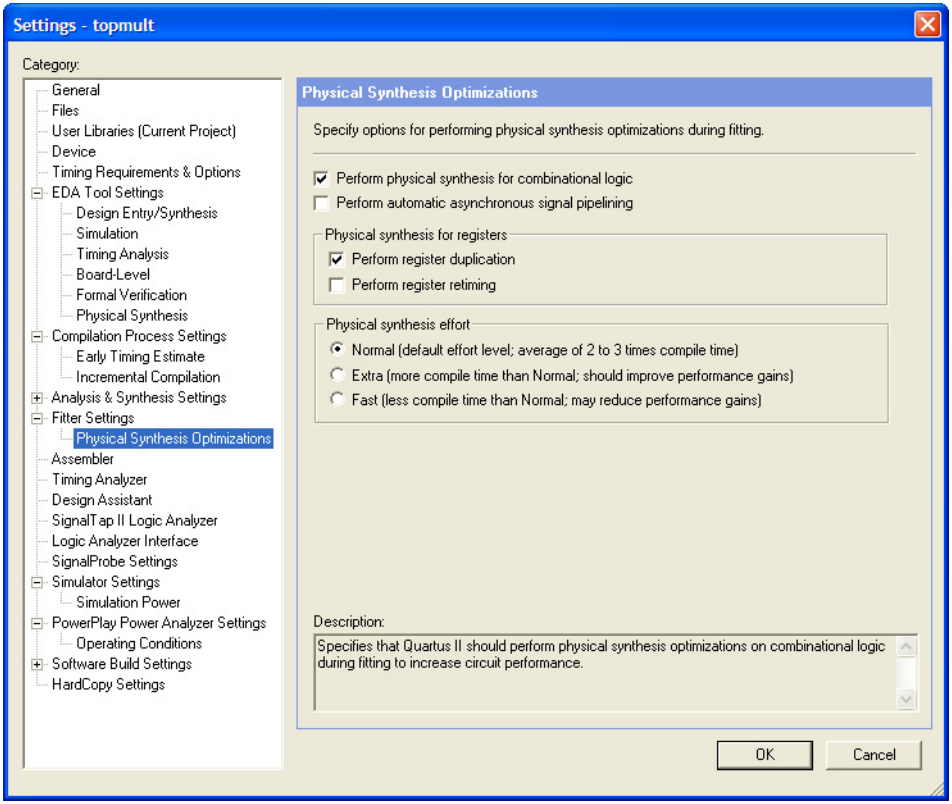
Figure 18–4. Synthesis Netlist Optimizations



12. In the Category list, click the + icon to expand **Fitter Settings** and select **Physical Synthesis Optimizations**. The **Physical Synthesis Optimizations** page is shown.

13. Turn off the **Perform register retiming** option (Figure 18–5).

Figure 18–5. Setting Parameters for Netlist Optimizations



Performing register retiming on a design usually results in moving and merging/duplicating registers along the critical path. Because equivalence checkers compare the cones of logic terminating at registers, you should not move or duplicate the registers during optimization by the Quartus II software. If the options in this section are not selected, the Formality software script could be presented with a different set of compare points, and the resulting netlist is difficult to compare against the reference netlist file.

The Quartus II software, beginning with version 4.2, supports register duplication to improve the timing by duplicating the logic.



To learn more about register duplication, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

14. Perform a full compilation of the design either on the Processing menu by clicking **Start Compilation** or by clicking on the start compilation arrow icon located in the tool bar.

Handling Black Boxes

Every design entity in the golden netlist must have a corresponding formal verification model in order to successfully run formal verification. Design entities in the golden netlist without a corresponding formal verification model are handled as black boxes whose boundary interfaces must be preserved. These design entities appear in the netlist if one of the following situations apply:

- Altera megafunctions including library of parameterized modules (LPM's)

  The black-box property is only applied to LPM modules that do not have a formal verification model.

- Encrypted intellectual property (IP) cores
- Entities that are defined in the design format other than Verilog HDL or VHDL

The Quartus II software has the capability of automatically identifying the black boxes and sets the property Preserve Hierarchical Boundary to Firm to preserve the boundary interfaces of the black boxes which helps the formal verification.

You can also specify the black box property on entities that should be compared by the Formality software. To do this make the following assignments either using Tcl commands or GUI for the entities in question:

Tcl Command

The following two commands preserves the boundary interface of the entity: dram.

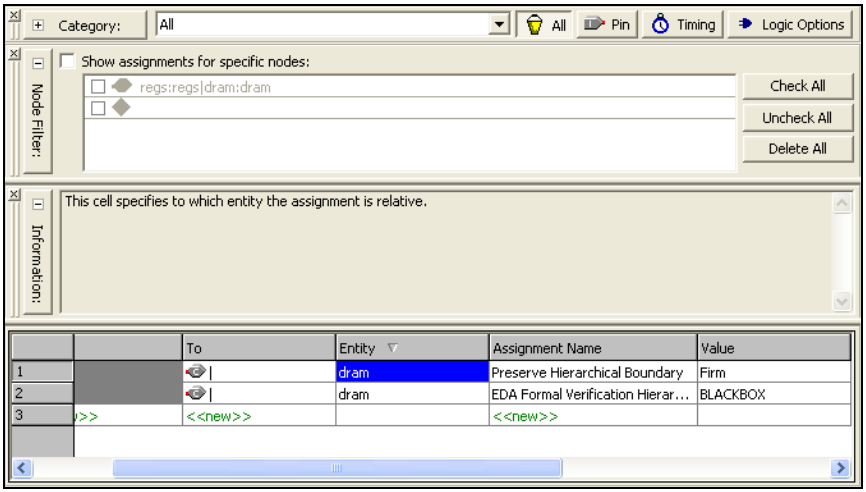
```
set_instance_assignment -name\
PRESERVE_HIERARCHICAL_BOUNDARY FIRM -to | -entity dram
set_instance_assignment -name EDA_FV_HIERARCHY\
BLACKBOX -to | -entity dram
```

GUI

Preserving the boundary interface of an entity using GUI.

- Assign the EDA Formal Verification Hierarchy value as blackbox.
- Assign the Preserve Hierarchical Boundary assignment with a value of Firm to the entity (Figure 18–6).

Figure 18–6. Making a Black Box Assignment to an Entity



The Quartus II software compiler generates the following files and directories:

- VO file: <*design_name*>**.vo.**
- Script file: <*design_name*>**.fms** used with Formality software.
- A black-box directory: black boxes that contains all the user defined black-box entities in the design which is located in the following directory: **/**<*project directory*>**/fv/formality/blackboxes**.

The script file contains the setup constraints used along with the Formality software. The <*entity*>**.v** file in the black-boxes directory contains the module description of only those entities that are not defined in the formal verification library.

For a sample script containing the setup commands generated by the Quartus II software, refer to "Tcl Sample Script" on page 18–13.

Quartus II Scripts for Formality

The Quartus II software generates scripts to use with the Formality software. This section describes the Formality software commands used within the scripts to help customers comparing the implementation and reference netlists. Table 18–2 describes the Formality software commands within Quartus II generated scripts.

Table 18–2. Formality Software Commands within Quartus II Generated Scripts

Command	Affect
`read_verilog` <*design files*>	This command reads both the reference and implementation netlists in addition to the appropriate library models.
`set_compare_rule` <*rule*>	Adds a name matching rule that Formality software applies to a design before creating compare points.
`set signature_analysis_matching` <*value*>	Use this command to specify whether or not to use signature analysis to match previously compared points.
`set_constant` <*value*>	This command allows you to set the logic state of a design object to either 0 or 1.
`set hdlin_altera_generate_naming` <*value*>	This command directs Formality software to apply alter naming conventions for registers.
`Set_user_match` <*mapping_point_name*>	Use this command to create pairs of matched points to compare those that Formality software could not match during its matching process.

Comparing Designs Using the Formality Software

To verify the functional equivalence between post-synthesis and post-place-and-route netlists, use the script file <*file_name*>**.fms** since it contains references to the macros defined in the Altera formal verification library. Some of the macros used are:

- _ALTERA_FAMILY_IS_STRATIX_
- POST_FIT
- FORMALITY
- GATES_TO_GATES

An example on the use of these macros is shown in the **read_verilog** command in the previous section. This script file <*file_name*>**.fms** is executed from either the GUI or using the following command:

```
%formality -file <file_name>.fms
```

For more information about using the Formality software, refer to the *Formality User Guide.*



The Formality software does not support inferred RAMs in RTL while performing RTL-to-Gates verification. Therefore, you should apply the black box property to RAM that is instantiated by the RTL code.

Known Issues and Limitations

This section discusses known issues and limitations of the formal verification flow using the DC FPGA, Quartus II, and Formality software:

1. Formal verification of post synthesis verses post-place-and-route netlist does not support latches because latches are implemented using combinational logic with a feedback loop which poses a problem to the Formality software.

2. If an LPM or an Altera megafunction module is inferred and all the ports of the module are not used, then unused ports should be connected to default values in the post-synthesis Verilog HDL netlist.

3. The Quartus II software may optimize away logic feeding a black box, resulting in mismatches on the blackbox inputs. For example, if certain bits of a RAM output are not being used, then the Quartus II software optimizes away the logic feeding the corresponding data inputs.

Conclusion

Formal verification enables verification of the design during all stages from RTL to place-and-route. As designs become larger, design verification using traditional methods becomes too time consuming. Thus, formal verification easily verifies that any modifications to the netlist in the physical domain have not altered from the Golden netlist. Advanced debugging capabilities within Formality software pinpoints the source of the differences between the Reference and Implementation netlists, enabling the user to easily fix the differences.

Related Links

Altera website: About Using the DC FPGA Software with the Quartus II Software

Tcl Sample Script

This section provides an example of the DC FPGA software script to perform synthesis and an example formal verification script generated by the Quartus II software.

DC FPGA Synthesis Script

The following example script presents the Altera recommended settings in the DC FPGA software for synthesizing the design for the Stratix architecture. The script also generates the Verilog netlist file for formal verification using the Formality software. These tasks are performed in the following five sections of the script:

- Setting up the library
- Default synthesis settings for Altera Stratix
- Analyzing the design files
- Compiling the design
- Generating the Verilog netlist for formal verification

```
# Setup file for Altera Stratix Devices
# Tcl style setup file but will work for
# original DC shell as well
# Need to define the root location of the
# libraries by changing
# the variable $dcfpga_lib_path
set dcfpga_lib_path "<dcfpga_rootdir>\
/libraries/fpga/altera"
set search_path ". $dcfpga_lib_path
$dcfpga_lib_path/STRATIX $search_path"
set target_library "stratix.db"
set synthetic_library "tmg.sldb altera_mf.sldb\
lpm.sldb"
set link_library "* stratix.db tmg.sldb\
altera_mf.sldb\ lpm.sldb"
set cache_dir_chmod_octal "1777"
set hdlin_enable_vpp "true"
set post_compile_cost_check "false"
set_fpga_defaults -formality altera_stratix
set formality_altera_debug true
set_verification_friendly_mode -filename
<top_level>.svf -append \
-allow_override
set verilogout_no_tri true
set verilogout_write_constant_nets true
set compile_fix_multiple_port_nets true
## Setup design directory for database, temporary files
# and netlist
#</OUTPUTDIR>#
set outputdir <directory_name>
file mkdir $outputdir/WORK
define_design_lib WORK -path $outputdir/WORK
```

```
## Setup the Top-level design name
set top <top_level_module>
##Setup synthesis optimization options
set dcfsm_force_encoding neutral
#<READFILES>#
##Analyze source files
##Elaborate design
elaborate $top
#</ELABORATE>#
##Specify Target device
current_design $top
set_fpga_target_device AUTOFASTEST
## Insert pad during synthesis
set_port_is_pad [get_ports "*"]
#<FPGACONST>#
## Specify clock constraints
#</FPGACONST>#
#<COMPILE>#
##Setup compile options
ungroup -small 500
## Compile design
compile
change_names -rule verilog -hierarchy
#<REPORT>#
##Generate netlist/reports/constraints for PAR
write -hier -f verilog -o $outputdir/$top.v
report_fpga > $outputdir/fpga.rpt
```

Quartus II Software-Generated Formal Verification Script

The following example script shows the sample setup commands generated by Quartus II software:

```
read_verilog -i -vcs \
"+define+_ALTERA_FAMILY_IS_STRATIX_ \
+define+POST_FIT \
+define+FORMALITY -y $QUARTUS/eda/fv_lib/verilog \
+libext+.v -y \
/home/formality/testcases/mult/quartus/fv/ \
formality/blackboxes" \
$PROJECT/fv/formality/mult_ram.vo
set_top mult_ram
set_black_box i:/WORK/altsyncram
report_black_box
set_compare_rule i:/WORK/mult_ram -from "_aI$" -to ""
set_compare_rule r:/WORK/mult_ram -from "\/" -to "_a"
set_compare_rule i:/WORK/mult_ram -from "\/" -to "_a"
match
verify
```

Referenced Documents

This chapter references the following documents:

- *Formality User Guide*
- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 18–3 shows the revision history for this chapter.

Table 18–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Reorganized "Referenced Documents" on page 18–15.	—
May 2007 v7.1.0	Added Referenced Documents.	—
March 2007 v7.0.0	Updated Quartus II software 7.0 revision and date only. No other changes made to chapter.	—
November 2006 v6.1.0	Added new revision history table format to the document.	—
May 2006 v6.0.0	Minor updates for the Quartus II software version 6.0.0.	—
October 2005 v5.1.0	• Updated for the Quartus II software version 5.1. • Chapter 15 was previously Chapter 13 in version 5.0.	—
May 2005 v5.0.0	New functionality for Quartus II software 5.0.	—
January 2005 v1.0	Initial release.	—

Section VII. Device Programming

The Quartus® II software offers a complete software solution for system designers who design with Altera® FPGA and CPLD devices. The Quartus II Programmer is part of the Quartus II software package that allows you to program Altera CPLD and configuration devices, and configure Altera FPGA devices. This section describes how you can use the Quartus II Programmer to program or configure your device after you successfully compile your design.

This section includes the following chapter:

- Chapter 19, Quartus II Programmer



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

19. Quartus II Programmer

QII53022-7.2.0

Introduction

The Quartus® II software offers a complete software solution for system designers who design with Altera® FPGA and CPLD devices. The Quartus II Programmer is part of the Quartus II software package that allows you to program Altera CPLD and configuration devices, and configure Altera FPGA devices. After your design successfully compiles, you can use the Quartus II Programmer to program or configure your device.

This chapter contains the following sections:

- "Programming Flow"
- "Programming and Configuration Modes" on page 19–4
- "Programmer Overview" on page 19–6
- "Hardware Setup" on page 19–12
- "Device Programming and Configuration" on page 19–14
- "Optional Programming Files" on page 19–18
- "Flash Loaders" on page 19–21
- "Other Programming Tools" on page 19–22
- "Scripting Support" on page 19–22

Programming Flow

The programming flow begins with design compilation, in which the Quartus II Assembler generates the programming or configuration file, then proceeds with the programming or configuration file conversion for different configuration devices, or optional programming and configuration file creation. The flow ends with the configuration or programming of the FPGA, CPLD, or configuration devices with the programming or configuration file using the Quartus II Programmer.

Figure 19–1 shows the programming file generation flow. This flow covers the types of configuration and programming files that are used by the Quartus II Programmer. Refer to "Optional Programming Files" on page 19–18 for information on other optional programming files.

Figure 19–1. Programming File Generation Flow

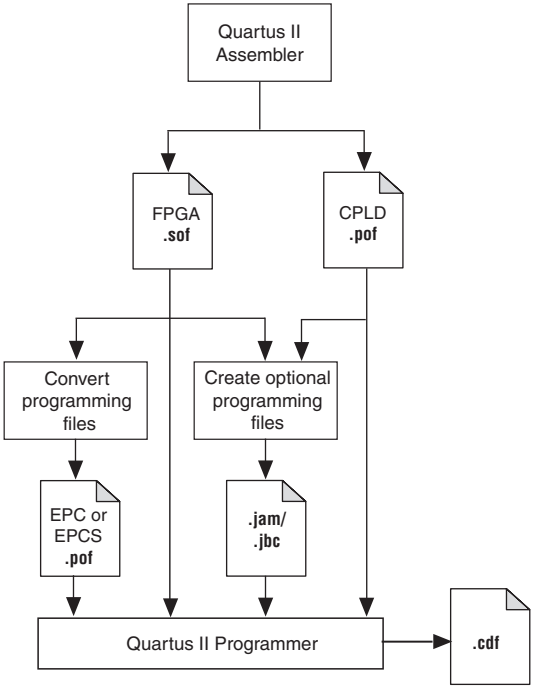


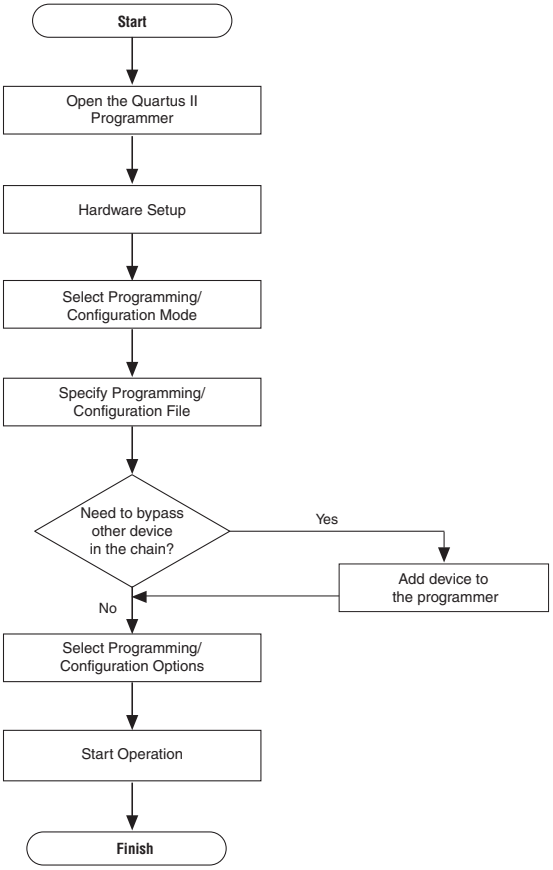
Table 19–1 shows the programming and configuration file formats supported by Altera FPGAs, CPLDs, configuration devices, enhanced configuration devices, and serial configuration devices.

Table 19–1. Programming and Configuration File Format

File Format	FPGA	CPLD	Configuration Device and Enhanced Configuration Device	Serial Configuration Device
SOF	✓	—	—	—
POF	—	✓	✓	✓
Jam	✓	✓	✓	—
JBC	✓	✓	✓	—

Figure 19–2 shows the programming flow using the Quartus II Programmer. Refer to "Generating Optional Programming Files" on page 19–20 for detailed information about converting or creating different programming files. Refer to "Device Programming and Configuration" on page 19–14 for information about programming or configuring the device.

Figure 19–2. Programming Flow



Programming and Configuration Modes

The Quartus II Programmer supports the following four programming or configuration modes: JTAG, passive serial, active serial, and in-socket programming.

JTAG Mode

You can use the Joint Test Action Group (JTAG) mode to configure FPGA devices and program CPLDs, configuration devices, or enhanced configuration devices. The JTAG mode allows multiple FPGAs, CPLDs, and configuration devices connected in a JTAG chain to be configured or programmed at the same time. JTAG programming or configuration uses four JTAG pins: TCK, TDI, TMS, and TDO. The JTAG interface also allows you to perform boundary-scan test using third-party boundary scan tools.

POF files are used for programming CPLDs, and configuration or enhanced configuration devices, while SOF files are used for configuring FPGA devices. Jam and JBC files can be used for both programming and configuration. Serial configuration devices do not support JTAG programming.



For more information about JTAG configuration or programming mode and JTAG pin connection, refer to the *Configuration Handbook*, or the device handbook or data sheet for the respective FPGA, CPLD, or configuration devices.

Passive Serial Mode

You can use the passive serial (PS) mode to configure Altera FPGAs. PS configuration uses the DCLK, CONF_DONE, nCONFIG, nSTATUS, and DATA0 configuration pins. Unlike the JTAG scheme, the PS configuration scheme can be used to configure the FPGA with a configuration device or enhanced configuration device, not necessarily through a download cable. If you are using the configuration device or enhanced configuration device to configure the FPGA through PS mode, you can route the configuration signals out to a header so that you can also configure the FPGA through the download cable with the Quartus II Programmer. Configuration through PS mode with a download cable is useful in the design stage. This configuration method allows you to configure your FPGA device directly from the Quartus II Programmer as you make changes to your design for debugging and testing.

PS mode supports configuration of an FPGA chain. SOF files are used for configuration through PS. Every FPGA device in the chain requires a SOF, so the number of SOF files used depends on the number of FPGA devices in the chain.

For more information about PS configuration mode and PS pin connection, refer to the *Configuration Handbook* or the chapter on configuration in the appropriate FPGA device handbook.

Active Serial Mode

You can use the active serial (AS) mode to program serial configuration devices. After programming completes, the serial configuration device then configures the FPGA. AS programming uses the DATA, DCLK, nCS, and ASDI pins. If the download cable is connected to the nCONFIG and nCE pins of the FPGA, the download cable disables the FPGA's access to the AS interface by holding the nCE pin high and the nCONFIG pin low. Upon completion of the programming, the nCE and nCONFIG pins are released and the FPGA configuration begins.

For more information about programming the serial configuration device, configuring the FPGA with the serial configuration device through AS mode, or the AS pin connection, refer to the *Serial Configuration Data Sheet* in the *Configuration Handbook* or the chapter on configuration in the appropriate FPGA device handbook.

In-Socket Programming Mode

The in-socket programming mode is used for programming a single device. This programming mode supports programming the MAX® 7000 and MAX 3000 CPLD families, configuration devices, enhanced configuration devices, and serial configuration devices. Instead of using a download cable, in-socket programming mode uses the Altera Programming Unit (APU) hardware together with the programming adapter for the corresponding device to program the device. The programming unit with the programming adapter has a socket for the device and the hardware powers the device for programming. In-socket programming is normally used in the production environment to pre-program devices before they are mounted on the printed circuit boards on the assembly line.

Refer to **www.altera.com** or the Quartus II Help for a list of programming adapters available for Altera devices.

Table 19–2 shows the programming and configuration modes supported by Altera devices.

Table 19–2. Programming and Configuration Modes

Mode	FPGA	CPLD	Configuration Device and Enhanced Configuration Device	Serial Configuration Device
JTAG	✓	✓	✓	—
PS	✓	—	—	—
AS	—	—	—	✓
In-Socket Programming	—	✓(1)	✓	✓

Note to Table 19–2:
(1) MAX II CPLDs do not support in-socket programming mode.

Programmer Overview

The Quartus II Programmer graphical user interface (GUI) is a window in which you can add your programming and configuration files, specify the programming options and hardware, and then proceed with the programming or configuration of the device.

To open the **Programmer** window, on the Tools menu, click **Programmer**. Figure 19–3 shows the programmer GUI. The status of each operation, whether programming is successful or not, is reported in the Quartus II message window. Figure 19–4 shows a typical programming message after the programmer has successfully programmed a device.

Figure 19–3. The Programmer Window

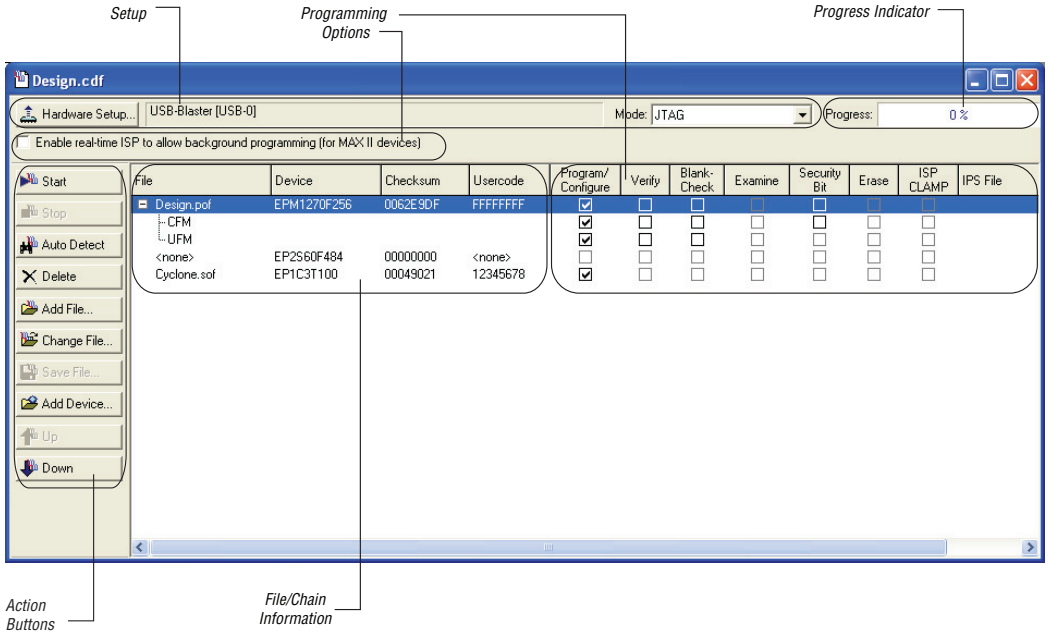


Figure 19–4. Status Report in the Message Window

Type	Message
	Info: Successfully performed operation(s)
	Info: Ended Programmer operation at Mon Feb 26 09:39:27 2007
	Info: Started Programmer operation at Mon Feb 26 09:39:51 2007
	Info: Device 1 contains JTAG ID code 0x020A30DD
	Info: Device 1 silicon ID is ALTERA04-1
	Info: Examining devices
	Info: Successfully performed operation(s)
	Info: Ended Programmer operation at Mon Feb 26 09:39:53 2007

Table 19–3 describes the items available in the programmer window.

Table 19–3. Programmer Window Items (Part 1 of 3)

Items	Description
Hardware Setup	Opens the **Hardware Setup** dialog box in the programmer and enables you to perform the following: • add and remove hardware items from the **Hardware** list. • add and remove JTAG servers from the **JTAG Servers** list. • configure your local JTAG server. • specify a programming hardware or download cable for device programming and configuration.
Mode	Specifies the programming or configuration mode (either JTAG, In-Socket Programming, Passive Serial, or Active Serial Programming).
Progress	Shows the progress of a specific operation.
Action Buttons	
Start	Starts the operations of the specified programming options.
Stop	Stops all operations in progress.
Auto Detect	Scans the JTAG chain to check for devices in the chain and the chain connection.
Delete	Removes the selected programming or configuration files from the programmer.
Add File	Adds programming or configuration files to the programmer.
Change File	Replaces the selected programming or configuration file with another file.
Save File	Allows you to save the data read out from CPLD or configuration devices using the “examine” process into a POF file.
Add Device	Adds a device into the JTAG device chain in the programmer. If no programming or configuration file is specified, the programmer will bypass this device during programming or configuration. You can also add your user-defined device into the chain.
Up	Moves the selection up to another programming or configuration file or device in the programmer window.
Down	Moves the selection down to another programming or configuration file or device in the programmer window.
File or Device Chain Information	
File	Displays the programming or configuration file name.
Device	The **Device** column shows: • the target device of the file, if you add a programming or configuration file into the programmer. • the devices in the JTAG chain detected by the programmer, if you click **Auto Detect** in JTAG mode. • the device added to the programmer, if you manually add a device into the programmer.

Table 19–3. Programmer Window Items (Part 2 of 3)

Items	Description
Checksum	The **Checksum** column shows: ● the checksum of the file, if you add a programming or configuration file into the programmer. ● the checksum for the data read out, if you examine a device. The checksum is calculated by the Quartus II software. The programmer does not display the checksum for the Jam or JBC files generated for a multi-device JTAG chain.
Usercode	The **Usercode** column shows: ● the usercode of the file, if you add a programming or configuration file into the programmer. ● the usercode read out from the device, if you examine a device. You can specify the usercode before design compilation, or use the Auto usercode feature that uses the checksum as the usercode. The programmer does not show the usercode information in PS configuration mode or for the Jam or JBC files generated for a multi-device JTAG chain.
Programming Options	
Enable real-time ISP to allow background programming	Can only be turned on if you are targeting a MAX II device, and is turned off for all other device families. When this option is turned on, you can do the real-time in-system programming (ISP) for the MAX II device. The existing design in the MAX II device functions normally during and after the real-time ISP is completed. The new design starts to function after a power cycle to the device occurs.
Program or Configure	Can be used for programming CPLDs, configuration devices, or configuring FPGA devices.
Verify	Verifies the content of the CPLD and all configuration devices against the respective programming files. This option is not available for FPGAs. Verification fails if the data in the file is different from the data in the device. Stand-alone verification for the CPLD with the programming file used for the programming will fail if the security bit is set when the device is programmed initially.
Blank-Check	Checks whether the CPLD or configuration device is blank or not.
Examine	Reads back the contents of the CPLD or configuration device. You can then save the examined data as a POF file. Examining a CPLD with the security bit set does not produce a usable POF file. MAX 7000S devices require you to add a valid MAX 7000S POF file that targets the same device before you can examine the data back from the device.
Security Bits	Protects the design in the CPLD from being examined. If the security bit is set when the CPLD is programmed, you cannot read the correct data out using the examine process. Security bits cannot be set for the configuration devices or FPGAs.

Table 19–3. Programmer Window Items (Part 3 of 3)	
Items	**Description**
Erase	Erases the contents of the CPLD and all configuration devices. You can also erase the user flash memory (UFM) of the MAX II CPLD. MAX 7000S devices require you to add a valid MAX 7000S POF file that targets the same device before you can erase the device.
ISP CLAMP	Allows the MAX II or MAX 7000B CPLD's I/O pins to be clamped to certain states during normal programming. ISP CLAMP can only be turned on if certain pins of the device have the ISP Clamp State assignment enabled, or you have added an I/O Pin State (IPS) file in the programmer.
IPS File	Shows the IPS file used for ISP Clamp of the MAX II or MAX 7000B CPLDs. The **IPS File** column only appears if your programmer window has a MAX II or MAX 7000B POF file. To add in the IPS file, click once on the row of the programming file and on the Edit menu, click **Add IPS File**.

Table 19–4 shows the programming and configuration options supported by Altera devices.

Table 19–4. Programming and Configuration Options				
Option	**FPGA**	**CPLD**	**Configuration Device and Enhanced Configuration Device**	**Serial Configuration Device**
Program or Configure	✓	✓	✓	✓
Verify	—	✓	✓	✓
Blank-Check	—	✓	✓	✓
Examine	—	✓	✓	✓
Security Bit	—	✓	—	—
Erase	—	✓	✓	✓
ISP Clamp	—	✓ *(1)*	—	—
IPS File *(2)*	—	✓	—	—
Real-time ISP	—	✓ *(3)*	—	—

Notes to Table 19–4:
(1) Only MAX II and MAX 7000B CPLDs support the ISP Clamp feature.
(2) IPS file is used for ISP Clamp.
(3) Only MAX II CPLDs support the real-time ISP feature.

Tools Menu

More programmer options are available from the Tools menu. On the Tools menu, click **Options** and then click **Programmer**. For descriptions of these options, refer to Table 19–5.

Table 19–5. Programmer Options

Option	Description
Show checksum without usercode	Determines whether the checksum values displayed in the programmer are calculated with or without JTAG usercodes. This option allows you to have multiple versions of a programming or configuration file with different user codes, but share the same checksum.
Initiate configuration after programming	Specifies that configuration devices configure attached FPGA devices automatically after the programmer completes programming the configuration devices.
Display message when programming finishes	Displays a message when programming or other operation such as examining or blank-checking is complete.
Enable real-time ISP to allow background programming (for MAX II devices)	Can only be turned on if you are targeting a MAX II device. This option is turned off for all other device families. When this option is turned on, you can do the real-time in-system programming (ISP) for the MAX II device. The existing design in the MAX II device functions normally during and after the real-time ISP is completed. The new design starts to function after a power cycle to the device occurs. This option is also available in the programmer window.
Halt on-chip configuration controller	Halts the on-chip auto-configuration controller of the FPGA device for AS configuration, or the configuration device for PS or Fast Passive Parallel (FPP) configuration to allow JTAG configuration through a download cable. If you want to configure your FPGA through JTAG while the FPGA `MSEL` pins are set to AS mode, you should halt the on-chip configuration controller if any of the following occurs: ● the active serial configuration device connected to your FPGA is blank ● the active serial configuration device is not present ● an error occurs during AS configuration prior to JTAG configuration If the `MSEL` pins are set to PS or FPP mode, halt the configuration controller of the configuration device if an error occurs during PS or FPP configuration prior to JTAG configuration. The FPGA pulls the `nSTATUS` pin (which is connected to the `OE` pin of the configuration device) low to disable the configuration device.
Automatically check the Program/Configure checkbox when adding SOF	Automatically enables the program or configuration operation when adding an SRAM Object File (**.sof**) to the file list in the programmer window.

Hardware Setup

The Quartus II Programmer provides the flexibility to choose the download cable or programming hardware. Before you can program or configure your device, you must have the correct hardware setup.

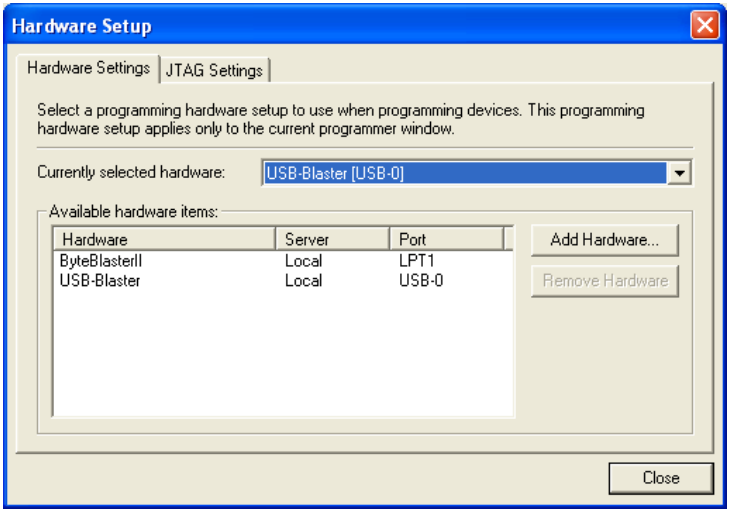
Hardware Settings

Click **Hardware Setup** to bring up the **Hardware Setup** dialog box. On the **Hardware Settings** tab (Figure 19–5), you can select a download cable or programming hardware available from the **Currently selected hardware** list. If the download cable or programming hardware you require is not displayed, click **Add Hardware** and specify the download cable or programming hardware. Make sure that you have installed the download cable driver before adding the hardware.



You do not need to manually add the USB-Blaster™ download cable to the list. After installing the driver, simply connect the download cable to the PC before opening the **Hardware Setup** dialog box. The USB-Blaster appears automatically in the list when the dialog box is opened.

Figure 19–5. Hardware Settings





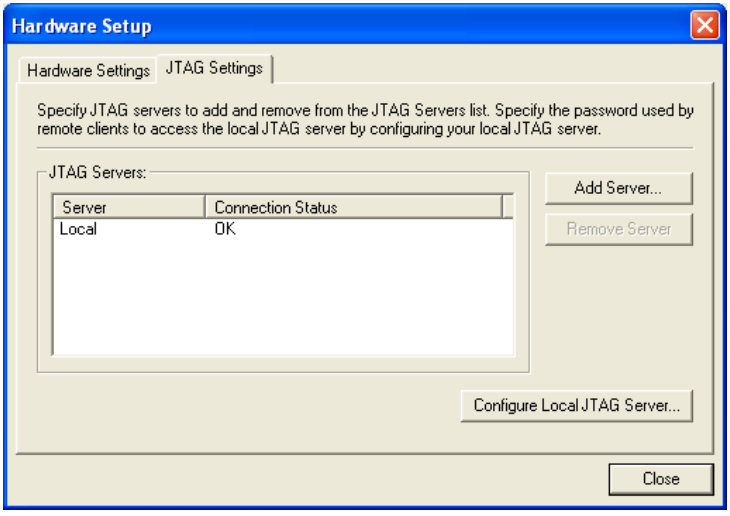
More information about programming hardware driver installation is available in the Design Software section under **Support** on the Altera website (**www.altera.com/support**).

JTAG Settings

The JTAG server allows programs such as the Quartus II Programmer to access the JTAG hardware. This application software is installed together with the Quartus II software. You can also access the JTAG download cable or programming hardware connected to a remote computer through the JTAG server of that computer. With the JTAG server, you can control the programming or configuration of devices from a single computer through other computers at remote locations. The JTAG server uses the TCP/IP communications protocol.

Click **Hardware Setup** to bring up the **Hardware Setup** dialog box. On the **JTAG Settings** tab (Figure 19–6), you can add or remove JTAG servers from the list. By default, you have only the local JTAG server (which is on your computer) in the list. By adding a remote JTAG server, you can access the JTAG hardware in that remote computer from your computer. You need the password of the remote JTAG server to add the server to your list. Click **Add Server**, then enter the IP address of that computer in the **Server name** field and the password in the **Server password** field.

Figure 19–6. JTAG Settings



You can also allow remote clients to access the JTAG server on your computer and program or configure devices connected to your computer through the JTAG interface of your computer. Click **Configure Local JTAG Server** to enable the server and then enter the password that the remote clients require to access your JTAG server.

Device Programming and Configuration

The Quartus II Programmer supports single- or multi-device programming and configuration. This section describes the steps required to program or configure Altera devices, as well as how to bypass Altera and non-Altera devices in a JTAG chain.

Single Device Programming and Configuration

To program or configure a single device with the Quartus II Programmer, perform the following steps:

1. On the Tools menu, click **Programmer** to open the **Programmer** window.
2. Click **Hardware Setup** and select the programming hardware or download cable. If you are using JTAG mode, you can specify the correct JTAG settings for programming or configuration involving remote JTAG servers. Click **Close**.
3. From the **Mode** list, select the programming or configuration mode.
4. Click **Add File** to add the POF or SOF file to the programmer (you can omit this step if the file is already displayed). To change the file, select it and click **Change File**. To remove the file from the programmer, select it and click **Delete**.



If you are using JTAG, AS, or in-socket programming mode, after the file has been added to the programmer, select the programming or configuration option by turning on the corresponding check box in the programmer.

5. Click **Start**.

Multi-Device Programming and Configuration

JTAG and PS modes allow you to program or configure a device chain. A JTAG chain can consist of a combination of FPGA, CPLD, and configuration devices that support JTAG mode. A PS chain consists of FPGAs that support PS mode. The steps for programming or configuring a device chain is similar to the steps for programming or configuring a single device. One exception is that in a device chain you must specify all the programming or configuration files for the devices you want to program or configure. JTAG mode allows you to bypass some of the devices in the JTAG chain while programming or configuring the rest of the devices. PS mode does not allow you to bypass devices in the FPGA chain.

Bypassing an Altera Device

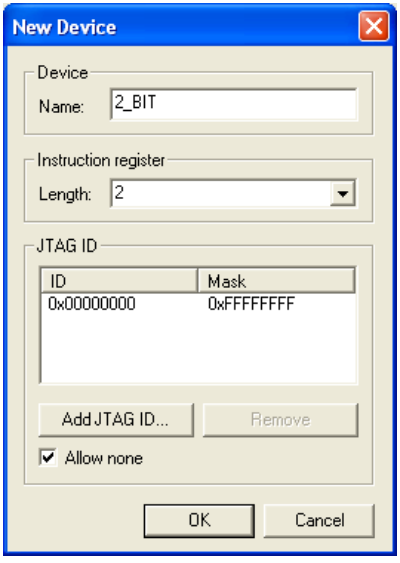
If you have the programming or configuration file for the Altera device you want to bypass, in the programmer, turn off all the options in the row for that device before you program or configure other devices. If you do not have the programming or configuration file for that device, click **Add Device** to specify the device.

Bypassing a Non-Altera Device

The JTAG chain of the device you want to program or configure may contain non-Altera devices. To program or configure your Altera device in the JTAG chain, you must bypass those non-Altera devices. The non-Altera devices are not in the list of devices that you can select when you click **Add Device** in the programmer.

To bypass the devices, you must manually define these devices. Click **Add Device** to open the **Select Device** dialog box. Click **New** to define a device. In the **New Device** dialog box (Figure 19–7), enter the name of the device and the JTAG instruction register length of the device. You can find the JTAG instruction register length in the device's data sheet. You can also specify the JTAG ID code for the device by clicking **Add JTAG ID**. This is optional and you can turn on **Allow none** to set the ID code to all 0s. If you do not specify the JTAG ID code, the default value is all 0s.

Figure 19–7. New Device Dialog Box



After defining the device, the device appears in the device list (Figure 19–8). Click **Export** to save the information in a Quartus User-Defined Device (QUD) file. This file saves the information for the user-defined devices that appear under **Device name** in the dialog box and can be used by other Quartus II projects as well. To obtain information on the user-defined devices from the QUD file, click **Import** and the devices are listed under **Device name**.

Figure 19–8. Select Devices

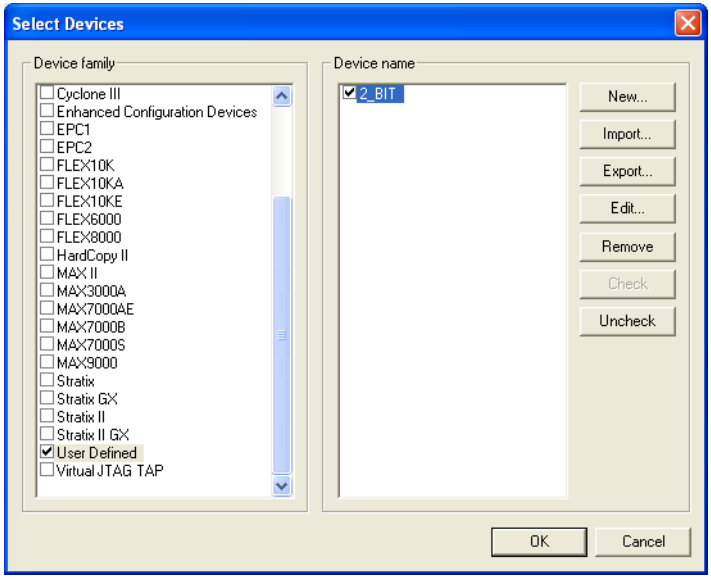
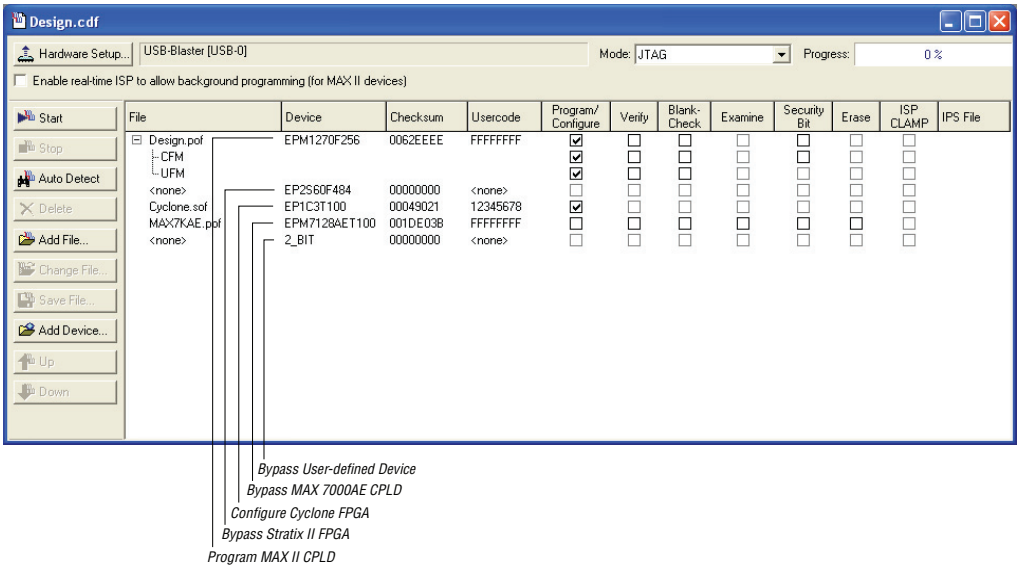


Figure 19–9 shows the programmer window for a JTAG chain.

Figure 19–9. Multi-Device JTAG Chain



Chain Description File

All the information in the Quartus II Programmer, including the programming or configuration mode, programming or configuration files used, device chain information, and the programming options specified can be saved in a chain description file (CDF). You do not have to specify the information each time you program the device chain. Simply open the CDF in the Quartus II software and information appears in the Quartus II Programmer GUI.

Design Security Key Programming

The Quartus II Programmer supports the generation of encryption key programming files and encrypted configuration files for Altera FPGAs that support the design security feature. You can also use the Quartus II Programmer to program the encryption key into the FPGA.

Refer to *AN 341: Using the Design Security Feature in Stratix II Devices* for more information about using the design security feature with the Quartus II software.

Optional Programming Files

The Quartus II software is able to generate optional programming or configuration files in various formats to be used with programming tools other than the Quartus II Programmer. In addition, you can convert the FPGA configuration files to programming files for configuration devices.

Types of Programming and Configuration Files

The Quartus II software generates programming files of various formats for use with different programming tools. Table 19–6 shows the programming and configuration files generated by the Quartus II software.

Table 19–6. Types of Programming and Configuration Files (Part 1 of 2)

File Format	Generated by the Quartus II Software	Supported by the Quartus II Programmer	Description
SOF	✓	✓	This configuration data file is used for configuring FPGA devices. The Quartus II Assembler generates this file when you compile your FPGA design.
POF	✓	✓	This programming data file is used for programming CPLDs and configuration devices. The Quartus II Assembler generates the CPLD POF file when you compile your CPLD design. The configuration device POF file is converted from the FPGA SOF file.
Jam	✓	✓	This ASCII-format file is used for configuring or programming one or more FPGAs, CPLDs, and configuration devices in a JTAG chain. The Jam file includes both programming algorithm and data. Apart from the Quartus II Programmer, you can use Altera's Jam Standard Test and Programming Language (STAPL) player, the **quartus_jli** executable, or other third-party programming tools together with the Jam file. The Jam file is also suitable for embedded processor-type programming environments.
JBC	✓	✓	Similar to the Jam file, this binary-format file is used for configuring or programming one or more FPGAs, CPLDs, and configuration devices in a JTAG chain. The JBC file includes both the programming algorithm and data, and the size is smaller than the Jam file. In addition to the Quartus II Programmer, you can use Altera's Jam Byte-Code player, the **quartus_jli** executable, or other third-party programming tools together with the JBC file. The JBC file is also suitable for embedded processor-type programming environments.

Table 19–6. Types of Programming and Configuration Files (Part 2 of 2)

File Format	Generated by the Quartus II Software	Supported by the Quartus II Programmer	Description
SVF	✓	—	This ASCII-format file is used for configuring, programming, blank-checking, and verifying one or more FPGAs, CPLDs, and configuration devices in a JTAG chain. The SVF file, which includes programming algorithm and data, is suitable for an automated test equipment (ATE) environment that requires a fixed programming algorithm.
ISC	✓	—	This data file is used with the IEEE 1532 BSDL file for programming a single device that supports IEEE 1532 programming. The Quartus II software supports generating the ISC file for MAX 7000AE, MAX 7000B, and MAX 3000A CPLDs.
Hexout	✓	—	The Hexout file is used for programming FPGA configuration data into enhanced configuration devices or other storage devices. For enhanced configuration devices, use the enhanced configuration device POF to generate the Hexout file. Use the FPGA SOF file to generate the Hexout file for other storage devices (for example, the flash or EEPROM devices). You can use a microcontroller to read back the data from the storage device and configure the FPGA. To program the enhanced configuration device or other storage devices with the Hexout file, you can use other third-party programming tools.
RBF	✓	—	This binary file contains configuration data for one or more FPGAs. You can use Altera's JRunner software to configure your FPGA device with the RBF file. The RBF file is also suitable for embedded processor configuration environments.
TTF	✓	—	This ASCII file contains configuration data for one or more FPGAs. The TTF file is used for embedded processor-type configuration.
RPD	✓	—	This binary file is used for programming serial configuration devices. Use the serial configuration device POF file to generate this file. You can use Altera's SRunner software to program your serial configuration device with the RPD file.
JIC	✓	✓	The JIC file is used for programming serial configuration devices through JTAG with the Quartus II Programmer and Altera FPGAs that support AS configuration mode.



Refer to the Quartus II Help or the *Configuration File Formats* chapter of the *Configuration Handbook* for more information about the programming and configuration file formats.



Refer to *AN 425: Using Command-Line Jam STAPL Solution for Device Programming* for more information about using the Jam and JBC programming files with the Jam STAPL Player, Jam STAPL Byte-Code Player, and the **quartus_jli** command-line executable.

Generating Optional Programming Files

When you compile your design, the Quartus II Assembler generates the SOF file for an FPGA or a POF file for a CPLD. With the SOF or POF for your design, you can then create other optional programming or configuration files, or convert the SOF to target a particular configuration device.

Create Programming Files

The Quartus II software allows you to create optional Jam, JBC, SVF, or ISC programming or configuration files. In addition, you can create Jam, JBC, and SVF files for a JTAG chain that consists of more than one device.

To create the files, open the Quartus II Programmer, set the programming or configuration mode to JTAG, and then add the programming or configuration files or devices to the programmer. On the File menu, click **Create/Update** and then click **Create JAM, SVF, or ISC File**. Select the file format and name the file accordingly.

For SVF files, you can create an SVF file for programming or verification only. In addition, you can specify whether or not to do the optional blank-check operation with the SVF file.

Convert Programming Files

To store the FPGA data into configuration devices, you can convert the SOF data to another format and program the configuration device. The Quartus II software supports converting the data into POF, Hexout, RBF, TTF, RPD, or JIC format.



For more information about converting programming files with the Quartus II software, refer to the *Configuration File Formats* chapter of the *Configuration Handbook*.

Generating Optional Programming or Configuration Files During Compilation

The Quartus II software can generate optional programming or configuration files automatically when you compile your design. To select the format of the optional programming or configuration files to be generated during compilation, on the Assignments menu, click **Settings**. Under **Device**, click **Device and Pin Options**.

You can select the configuration device from the **Configuration** tab for the configuration device POF generation. For other optional programming and configuration file generation, you can select the file format under the **Programming Files** tab.

Flash Loaders

Serial configuration devices and the common flash interface (CFI) flash devices do not support JTAG interface and cannot be programmed directly through the normal JTAG programming. Flash loaders allow the programming of the serial configuration device and the CFI flash from the Quartus II Programmer through JTAG.

Parallel Flash Loader

The parallel flash loader (PFL) performs two functions:

- Allows the programming of the CFI flash through the JTAG interface
- Acts as the configuration controller that reads the configuration data from the CFI flash and configures the FPGA

To program the CFI flash, the PFL uses the MAX II device as a bridge between the JTAG interface of the Quartus II Programmer and the CFI of the CFI flash device. You can program FPGA configuration data and user data into the flash with a flash POF generated by the Quartus II software. After the flash is programmed with the FPGA configuration data, the PFL is then used to read the configuration data back from the CFI flash to configure the FPGA.

Refer to *AN 386: Using the MAX II Parallel Flash Loader with the Quartus II Software* for more information about PFL.

Serial Flash Loader

The serial flash loader (SFL) allows programming of the serial configuration devices through JTAG. The SFL uses the FPGA device that supports AS configuration mode as a bridge between the active serial memory interface (ASMI) of the serial configuration device and the JTAG

interface of the programmer. The Quartus II Programmer uses the JIC file converted from the FPGA SOF file to program the serial configuration device though JTAG.



Refer to *AN 370: Using the Serial Flash Loader with the Quartus II Software* for more information about SFL.

Other Programming Tools

This section covers other programming tools that are related to the Quartus II Programmer and can be used for programming or debugging programming problems.

Quartus II Stand-Alone Programmer

If you do not have the full version of the Quartus II software, Altera offers the free Quartus II Stand-Alone Programmer. This stand-alone programmer has the full function of the normal Quartus II Programmer, and enables you to create or convert programming files from the SOF or POF of your design. You can download the Quartus II Stand-Alone Programmer from the **Download Center** page found through the **Support** page on the Altera website at **www.altera.com**.

jtagconfig Debugging Tool

The **jtagconfig** command-line utility is included with the Quartus II software. You can use this utility (which is similar to the auto detect operation in the Quartus II Programmer) to check the devices in a JTAG chain and the user-defined devices.

For more information about the **jtagconfig** utility, type one of the following commands at the command prompt:

```
jtagconfig -h ↵
jtagconfig --help ↵
```

Scripting Support

Apart from the Quartus II Programmer GUI, you can perform programming with the Quartus II command-line programmer (**quartus_pgm**). This **quartus_pgm** command-line programmer comes with the Quartus II Programmer. You can run this programmer separately from the Quartus II software. You can also run the procedures for the programmer in a Tcl script. The programmer accepts the POF, SOF, and JIC programming or configuration files. You can also use the CDF.

For more information about the command-line syntax, type one of the following commands at the command prompt:

```
quartus_pgm -h ↵
quartus_pgm --help ↵
```

For more information about a specific programmer option or topic, type the following command at the command prompt:

```
quartus_pgm --help=<option|topic> ↵
```

The following is an example of a command that programs a device:

```
quartus_pgm -c byteblasterII -m jtag -o bpv;design.pof ↵
```

where:

`-c byteblasterII` specifies the ByteBlaster II download cable
`-m jtag` specifies the JTAG programming mode
`-o bpv` represents the blank-check, program, and verify operations
`design.pof` represents the POF file used for the programming

The programmer automatically executes the erase operation before programming the device.

For detailed information about scripting command options, you can also refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Scripting Reference Manual* includes the same information in PDF format.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information on all settings and constraints in the Quartus II software. Refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook* for more information about command-line scripting.

Conclusion

The Quartus II Programmer offers you a wide variety of options to program and configure your Altera devices. With the Quartus II Programmer, the Quartus II software provides you with a complete solution for your FPGA or CPLD design prototyping, which can even be performed in the production environment.

Referenced Documents

This chapter references the following documents:

- *AN 341: Using the Design Security Feature in Stratix II and Stratix II GX Devices*
- *AN 370: Using the Serial FlashLoader with the Quartus II Software*
- *AN 386: Using the MAX II Parallel Flash Loader with the Quartus II Software*
- *AN 425: Using Command-Line Jam STAPL Solution for Device Programming*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Configuration File Formats* chapter of the *Configuration Handbook*
- *Configuration Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Reference Manual*
- *Serial Configuration Devices (EPCS1, EPCS4, EPCS16, and EPCS64) and EPCS128) Data Sheet* of the *Configuration Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 19–7 shows the revision history for this chapter.

Table 19–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007, v7.2.0	Reorganized "Referenced Documents".	—
May 2007 v7.1.0	Initial release	—



Quartus II Version 7.2 Handbook

Volume 4: SOPC Builder

101 Innovation Drive
San Jose, CA 95134
www.altera.com

QII5V4-7.2

Copyright © 2007 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

I.S. EN ISO 9001

Contents

Chapter Revision Dates ..... **xi**

About this Handbook ..... **xiii**
How to Contact Altera xiii
Typographic Conventions xiii

Section I. SOPC Builder Features

Chapter 1. Introduction to SOPC Builder
Overview 1–1
Architecture of SOPC Builder Systems 1–2
SOPC Builder Components 1–2
Example System 1–3
Custom Components 1–4
System Interconnect Fabric 1–5
Functions of SOPC Builder 1–5
Defining and Generating the System Hardware 1–5
Creating a Memory Map for Software Development 1–6
Creating a Simulation Model and Test Bench 1–6
Getting Started 1–7
Referenced Documents 1–7
Document Revision History 1–8

Chapter 2. System Interconnect Fabric for Memory-Mapped Interfaces
Introduction 2–1
High-Level Description 2–1
Fundamentals of Implementation 2–4
Functions of System Interconnect Fabric 2–4
Address Decoding 2–5
Datapath Multiplexing 2–6
Wait-State Insertion 2–7
Pipeline Read Transfers 2–8
Native Address Alignment and Dynamic Bus Sizing 2–9
Dynamic Bus Sizing 2–9
Wider Master 2–10
Narrower Master 2–10
Native Address Alignment 2–11
Arbitration for Multimaster Systems 2–12

Traditional Shared Bus Architectures 2–12
Slave-Side Arbitration 2–13
Arbiter Details 2–14
Arbitration Rules 2–15
Setting Arbitration Parameters in SOPC Builder 2–15
Fairness-Based Shares 2–16
Round-Robin Scheduling 2–17
Burst Transfers 2–17
Minimum Share Value 2–17
Burst Management 2–18
Clock Domain Crossing 2–19
Description of Clock Domain-Crossing Logic 2–19
Location of Clock Domain Crossing Logic 2–21
Duration of Transfers Crossing Clock Domains 2–22
Implementing Multiple Clock Domains in SOPC Builder 2–22
Component Overview 2–23
Functional Description 2–23
Interfaces 2–24
Clock Domain Crossing Logic and FIFOs 2–24
Burst Support 2–25
Example System with Avalon-MM Clock-Crossing Bridges 2–26
Instantiating the Avalon-MM Clock-Crossing Bridge in SOPC Builder 2–28
Interrupts 2–29
Software Priority 2–29
Hardware Priority 2–30
Assigning IRQs in SOPC Builder 2–30
Reset Distribution 2–31
Referenced Documents 2–31
Document Revision History 2–32

Chapter 3. System Interconnect Fabric for Streaming Interfaces

Introduction 3–1
High-Level Description 3–1
Avalon Streaming and Avalon Memory-Mapped Interfaces 3–2
Adapters 3–3
Data Format Adapter 3–4
Timing Adapter 3–4
Channel Adapter 3–5
Multiplexer Examples 3–5
Example to Double Clock Frequency 3–5
Example to Double Data Width and Maintain Frequency 3–6
Example to Boost the Frequency 3–6
Referenced Documents 3–7
Document Revision History 3–7

Chapter 4. SOPC Builder Components

Introduction 4–1

New Component Structure in v7.1 of the Quartus II Software 4–1
Component Providers 4–2
Component Hardware Structure 4–2
Components That Include Logic Inside the System Module 4–3
Components That Interface to Logic Outside the System Module 4–4
List of Available Components in SOPC Builder 4–4
Tcl Components 4–5
Component Description File (_hw.tcl) 4–5
Component File Organization 4–5
Referenced Document 4–6
Document Revision History 4–6

Chapter 5. Component Editor

Introduction 5–1
Component Hardware Structure 5–2
Starting the Component Editor 5–2
HDL Files Tab 5–2
Signals Tab 5–3
Naming Signals for Automatic Type and Interface Recognition 5–4
Templates for Interfaces to External Logic 5–5
Interfaces Tab 5–6
Component Wizard Tab 5–6
Identifying Information 5–6
Parameters 5–7
Saving a Component 5–7
Editing a Component 5–8
Referenced Documents 5–8
Document Revision History 5–9

Chapter 6. Building a Component Interface with Tcl Scripting Commands

Organization of a Component Tcl File 6–2
Set and Add Commands 6–3
Module Properties 6–4
Clock Interface 6–4
Avalon-MM Master Interface 6–5
Avalon-MM Slave Interface 6–5
Avalon-ST Source Interface 6–6
Avalon-ST Sink Interface 6–7
Avalon-MM Tristate Interface 6–7
Nios II Custom Instruction Interface 6–8
Interrupt Interface 6–9
Conduit Interface 6–10
Document Revision History 6–10

Chapter 7. Archiving SOPC Builder Projects

Introduction 7–1
Scope 7–1

Required Files 7–2
SOPC Builder Design Files 7–3
Nios II Application Software Project Files 7–3
Nios II System Library Project 7–4
File Write Permissions 7–4
Referenced Documents 7–4
Document Revision History 7–5

Section II. Building Systems with SOPC Builder

Chapter 8. Building Memory Subsystems Using SOPC Builder

Introduction 8–1
Example Design 8–2
Example Design Structure 8–2
Example Design Starting Point 8–4
Hardware and Software Requirements 8–5
Design Flow 8–5
Component-Level Design in SOPC Builder 8–6
SOPC Builder System-Level Design 8–6
Simulation 8–7
Quartus II Project-Level Design 8–7
Board-Level Design 8–7
Simulation Considerations 8–7
Generic Memory Models 8–7
Vendor-Specific Memory Models 8–8
On-Chip RAM and ROM 8–8
Component-Level Design for On-Chip Memory 8–8
Memory Type 8–8
Size 8–9
Read Latency 8–9
Non-Default Memory Initialization 8–9
Enable In-System Memory Content Editor Feature 8–10
SOPC Builder System-Level Design for On-Chip Memory 8–10
Simulation for On-Chip Memory 8–10
Quartus II Project-Level Design for On-Chip Memory 8–10
Board-Level Design for On-Chip Memory 8–11
Example Design with On-Chip Memory 8–11
EPCS Serial Configuration Device 8–12
Component-Level Design for an EPCS Device 8–12
SOPC Builder System-Level Design for an EPCS Device 8–12
Simulation for an EPCS Device 8–13
Quartus II Project-Level Design for an EPCS Device 8–13
Board-Level Design for an EPCS Device 8–13
Example Design with an EPCS Device 8–13
SDRAM 8–14

Component-Level Design for SDRAM 8–15
SOPC Builder System-Level Design for SDRAM 8–15
Simulation for SDRAM 8–15
Quartus II Project-Level Design for SDRAM 8–15
Connecting and Assigning the SDRAM-Related Pins 8–16
Accommodating Clock Skew 8–16
Board-Level Design for SDRAM 8–16
Example Design with SDRAM 8–16
Off-Chip SRAM and Flash Memory 8–19
Component-Level Design for SRAM and Flash Memory 8–20
Avalon-MM Tristate Bridge 8–21
Flash Memory 8–21
SRAM 8–22
SOPC Builder System-Level Design for SRAM and Flash Memory 8–22
Simulation for SRAM and Flash Memory 8–23
Quartus II Project-Level Design for SRAM and Flash Memory 8–23
Board-Level Design for SRAM and Flash Memory 8–24
Aligning the Least-Significant Address Bits 8–24
Aligning the Most-Significant Address Bits 8–25
Example Design with SRAM and Flash Memory 8–25
Adding the Avalon-MM Tristate Bridge 8–26
Adding the Flash Memory Interface 8–26
Adding the SRAM Interface 8–26
Adding the PLL 8–29
SOPC Builder System Contents Tab 8–30
Connecting and Assigning Pins in the Quartus II Project 8–31
Connecting FPGA Pins to Devices on the Board 8–33
Referenced Documents 8–34
Document Revision History 8–34

Chapter 9. Developing Components for SOPC Builder

Introduction 9–1
SOPC Builder Components and the Component Editor 9–1
Prerequisites 9–2
Hardware and Software Requirements 9–2
Component Development Flow 9–3
Typical Design Steps 9–3
Hardware Design 9–4
Software Design 9–6
Verifying the Component 9–8
Unit Verification 9–8
System-Level Verification 9–8
Design Example: Checksum Master 9–9
Install the Design Files 9–9
Review the Example Design Specifications 9–10
Checksum Design Files 9–11
Master Task Logic 9–11

Register File 9–12
Avalon-MM Clock Interface 9–12
Avalon-MM Master Interface 9–13
Avalon-MM Slave Interface 9–13
Software API 9–14
Create an SOPC Builder component 9–14
Open the Quartus II Project and Start the Component Editor 9–14
HDL Files Tab 9–15
Signals Tab 9–15
Interfaces Tab 9–18
Component Wizard Tab 9–23
Save the Component 9–23
Instantiate the Component in Hardware 9–24
Add the checksum Master Component to the SOPC Builder System 9–24
Compile the Hardware Design and Download to the Target Board 9–25
Exercise the Hardware Using Nios II Software 9–25
Start the Nios II IDE and Create a New IDE Project 9–26
Compile the Software Project and Run on the Target Board 9–28
Sharing Components 9–29
Referenced Documents 9–31
Document Revision History 9–31

Section III. Interconnect Components

Chapter 10. Avalon Memory-Mapped Bridges

Introduction to Bridges 10–1
Structure of a Bridge 10–1
Reasons for Using a Bridge 10–3
Address Mapping for Systems with Avalon-MM Bridges 10–7
Tools for Visualizing the Address Map 10–8
Differences between Avalon-MM Bridges and Avalon-MM Tristate Bridges 10–8
Avalon-MM Pipeline Bridge 10–9
Component Overview 10–9
Functional Description 10–10
The following sections describe the component's hardware functionality. 10–11
Interfaces 10–11
Pipeline Stages and Effects on Latency 10–11
Burst Support 10–12
Example System with Avalon-MM Pipeline Bridges 10–12
Instantiating the Avalon-MM Pipeline Bridge in SOPC Builder 10–13
Device Support 10–14
Installation and Licensing 10–14
Hardware Simulation Considerations 10–15
Software Programming Model 10–15
Referenced Documents 10–15

Document Revision History 10–15

Chapter 11. Avalon Streaming Interconnect Components

Introduction to Interconnect Components 11–1
Interconnect Component Usage 11–1
Address Mapping 11–3
Timing Adapter 11–3
Resource Usage and Performance 11–4
Instantiating the Timing Adapter in SOPC Builder 11–5
Input Interface Parameters 11–5
Output Interface Parameters 11–5
Common to Input and Output Interfaces 11–5
Channel Signal Width (Bits) 11–6
Max Channel 11–6
Bits Per Symbol 11–6
Symbols Per Beat 11–6
Include Packet Support 11–6
Error Signal Width (Bits) 11–6
Data Format Adapter 11–6
Resource Usage and Performance 11–7
Instantiating the Data Format Adapter in SOPC Builder 11–9
Input Interface Parameters 11–9
Data Symbols Per Beat 11–9
Output Interface Parameters 11–9
Data Symbols Per Beat 11–9
Common to Input and Output 11–9
Support Backpressure with the Ready Signal 11–9
Data Bits Per Symbol 11–9
Channel Signal Width (Bits) 11–9
Max Channel 11–9
Include Packet Support 11–10
Error Signal Width (Bits) 11–10
Channel Adapter 11–10
Resource Usage and Performance 11–10
Instantiating the Channel Adapter in SOPC Builder 11–11
Input Interface Parameters 11–11
Channel Signal Width (Bits) 11–11
Max Channel 11–11
Output Interface Parameters 11–11
Channel Signal Width (Bits) 11–11
Max Channel 11–11
Common to Input and Output Interfaces 11–11
Data Bits Per Symbol 11–11
Symbols Per Beat 11–12
Include Packet Support 11–12
Error Signal Width (Bits) 11–12
Device Support 11–12

Installation and Licensing 11–13
Hardware Simulation Considerations 11–13
Software Programming Model 11–13
Referenced Documents 11–13
Document Revision History 11–13

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook, Volume 4*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1. Introduction to SOPC Builder
Revised: *October 2007*
Part number: *QII54001-7.2.0*

Chapter 2. System Interconnect Fabric for Memory-Mapped Interfaces
Revised: *October 2007*
Part number: *QII54003-7.2.0*

Chapter 3. System Interconnect Fabric for Streaming Interfaces
Revised: *October 2007*
Part number: *QII54019-7.2.0*

Chapter 4. SOPC Builder Components
Revised: *October 2007*
Part number: *QII54004-7.2.0*

Chapter 5. Component Editor
Revised: *October 2007*
Part number: *QII54005-7.2.0*

Chapter 6. Building a Component Interface with Tcl Scripting Commands
Revised: *October 2007*
Part number: *QII54022-7.2.0*

Chapter 7. Archiving SOPC Builder Projects
Revised: *October 2007*
Part number: *QII54017-7.2.0*

Chapter 8. Building Memory Subsystems Using SOPC Builder
Revised: *October 2007*
Part number: *QII54006-7.2.0*

Chapter 9. Developing Components for SOPC Builder
Revised: *October 2007*
Part number: *QII54007-7.2.1*

Chapter 10. Avalon Memory-Mapped Bridges
Revised: *October 2007*
Part number: *QII54020-7.2.0*

Chapter 11. Avalon Streaming Interconnect Components
Revised: *October 2007*
Part number: *QII54021-7.2.0*

About this Handbook

This handbook provides comprehensive information about the Altera® SOPC Builder tool.

How to Contact Altera

For the most up-to-date information about Altera products, refer to the following table.

Information Type	USA and Canada
Technical support	www.altera.com/mysupport/
Technical training	www.altera.com/training/ custrain@altera.com
Product literature	www.altera.com/literature
Altera literature services	literature@altera.com
FTP site	ftp.altera.com

Typographic Conventions

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
“Subheading Title”	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: “Typographic Conventions.”

Visual Cue	Meaning
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input.` Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.

Section I. SOPC Builder Features

Section I of this volume introduces the SOPC Builder system integration tool, and describes the main features of the SOPC Builder tool. Chapters in this section serve to answer the following questions:

- What is SOPC Builder?
- What features does SOPC Builder provide?

This section includes the following chapters:

- Chapter 1, Introduction to SOPC Builder
- Chapter 2, System Interconnect Fabric for Memory-Mapped Interfaces
- Chapter 3, System Interconnect Fabric for Streaming Interfaces
- Chapter 4, SOPC Builder Components
- Chapter 5, Component Editor
- Chapter 6, Building a Component Interface with Tcl Scripting Commands
- Chapter 7, Archiving SOPC Builder Projects



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

1. Introduction to SOPC Builder

QII54001-7.2.0

Overview

SOPC Builder is a powerful system development tool for creating systems including processors, peripherals, and memories. SOPC Builder enables you to define and generate a complete system-on-a-programmable-chip (SOPC) in much less time than using traditional, manual integration methods. SOPC Builder is included in the Quartus® II software.

Many designers already know SOPC Builder as the tool for creating systems based on the Nios® II processor. However, SOPC Builder is more than a Nios II system builder; it is a general-purpose tool for creating systems that may or may not contain a processor.

SOPC Builder automates the task of integrating hardware components into a larger system. Using traditional system-on-chip (SOC) design methods, you must manually write top-level HDL files that wire together the pieces of the system. Using SOPC Builder, you specify the system components in a GUI, and SOPC Builder generates the interconnect logic automatically. SOPC Builder outputs HDL files that define all components of the system, and a top-level HDL design file that connects all the components together. SOPC Builder generates both Verilog HDL and VHDL equally, and does not favor one over the other. This chapter includes the following sections:

- "Architecture of SOPC Builder Systems" on page 1–2
- "Functions of SOPC Builder" on page 1–5
- "Getting Started" on page 1–7

In addition to its role as a system generation tool, SOPC Builder provides features to ease writing software and to accelerate system simulation.

This chapter introduces you to the architectural structure of systems built with SOPC Builder, and describes the primary functions of SOPC Builder.

Architecture of SOPC Builder Systems

This section describes the fundamental architecture of an SOPC Builder system.

An SOPC Builder component is a design module that SOPC Builder recognizes and can automatically integrate into a system. You can also define and add custom components. SOPC Builder connects multiple components together to create a top-level HDL file called the system module. SOPC Builder generates system interconnect fabric that contains logic to manage the connectivity of all components in the system.

SOPC Builder Components

SOPC Builder components are the building blocks for creating an SOPC Builder system. SOPC Builder components use Avalon® interfaces for the physical connection of components, and you can use SOPC Builder to connect any logical device (either on-chip or off-chip) that has an Avalon interface. There are two different Avalon interfaces:

- The Avalon® Memory-Mapped (Avalon-MM) interface uses an address-mapped read/write protocol that enables flexible topologies for connecting master components to read and/or write any slave components.
- The Avalon Streaming (Avalon-ST) interface is a high-speed, unidirectional, system interconnect that enables point-to-point connections between streaming components that send and receive data using source and sink ports.

SOPC builder components can use either Avalon-MM or Avalon-ST interfaces or both.

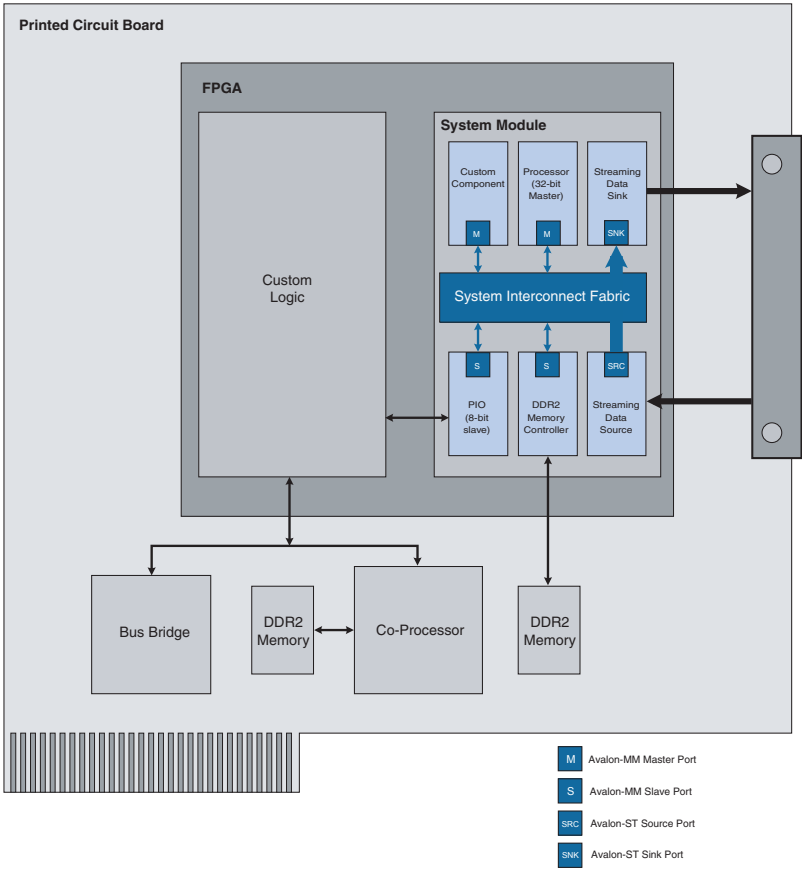


For details on the Avalon-MM interface, refer to the *Avalon Memory-Mapped Interface Specification* chapter in volume 4 of the *Quartus II Handbook*. For details about the Avalon-ST interface, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*. For details about the Avalon-ST interface protocol, refer to *Avalon Streaming Interface Specification*. All are available at **www.altera.com**.

Example System

Figure 1–1 shows an FPGA design including an SOPC Builder system module and custom logic modules. You can integrate custom logic inside or outside the system module. In this example, the custom component inside the system module is an SOPC Builder component that communicates with other modules through an Avalon-MM master interface. The custom logic outside of the system module is connected to the system module through a PIO interface. The system module includes two SOPC Builder components with Avalon-ST source and sink interfaces. The system interconnect fabric connects all of the SOPC Builder components using the Avalon-MM or Avalon-ST system interconnect as appropriate.

Figure 1–1. Example of an FPGA with a System Module Generated by SOPC Builder



A component can be a logical device that is entirely contained within the system module, such as the processor component shown in Figure 1–1. Alternately, a component can act as an interface to an off-chip device, such as the DDR2 interface component in Figure 1–1. In addition to the Avalon interface, a component can have other signals that connect to logic outside the system module. Non-Avalon signals can provide a special-purpose interface to the system module, such as the PIO in Figure 1–1. A component can be instantiated more than once per design.

Altera and third-party developers provide ready-to-use SOPC Builder components, including:

- Microprocessors, such as the Nios II processor
- Microcontroller peripherals, such as a scatter-gather DMA controller
- Timers
- Serial communication interfaces, such as a UART and a serial peripheral interface (SPI)
- General purpose I/O
- Digital signal processing (DSP) functions
- Communications peripherals, such as a 10/100/1000 Ethernet MAC
- Interfaces to off-chip devices, such as:
  - Buses and bridges
  - Application-specific standard products (ASSP)
  - Application-specific integrated circuits (ASIC)
  - Processors

Custom Components

SOPC Builder provides an easy method for you to develop and connect your own components. Your components can use either the Avalon-MM or Avalon-ST interfaces, or both. With the Avalon-MM interface, custom logic need only adhere to a simple interface based on address, data, read-enable, and write-enable signals. With the Avalon-ST interface, custom logic follows the configurable Avalon-ST interface protocol.

You use the following design flow to integrate custom logic into an SOPC Builder system:

1. Define the interface to the custom component.
2. Write HDL files describing the component in either Verilog HDL or VHDL.
3. Use the SOPC Builder component editor wizard to specify the interface and optionally package your HDL files into an SOPC Builder component.

4. Instantiate your component in the same manner as other SOPC Builder components.

Once you have created an SOPC Builder component, you can reuse the component in other SOPC Builder systems, and share the component with other design teams.



For instructions on developing a custom SOPC Builder component, refer to the *Developing SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*. For complete details about the file structure of a component, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*. For details about the SOPC Builder component editor, refer to the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*.

System Interconnect Fabric

The system interconnect fabric connects the components in SOPC Builder-generated systems. For Avalon-MM components, the system interconnect fabric is the collection of signals and logic that connects master and slave components, including address decoding, data-path multiplexing, wait-state generation, arbitration, interrupt controller, and data-width matching. For Avalon-ST components, the system interconnect fabric creates point-to-point connections between streaming components that send and receive data using source and sink ports.



For further details, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces* and *System Interconnect Fabric for Streaming Interfaces* chapters in volume 4 of the *Quartus II Handbook*.

Functions of SOPC Builder

This section describes the fundamental functions of SOPC Builder.

Defining and Generating the System Hardware

The purpose of SOPC Builder is to allow you to easily define the structure of a hardware system, and then generate the system. The GUI allows you to add components to a system, configure the components, and specify how they connect together.

After you add all components and system parameters, SOPC Builder generates the system interconnect fabric and output HDL files. During system generation, SOPC Builder outputs the following items:

- An HDL file for the top-level system module and for each component in the system
- A Block Symbol File (.**bsf**) representation of the top-level system module for use in Quartus II Block Diagram Files (**.bdf**)
- Software files for embedded software development, such as a memory-map header file and component driver
- (Optional) Testbench for the system module and ModelSim® simulation project files

After you generate the system module, you can compile it with the Quartus II software, or you can instantiate it in a larger FPGA design.

Creating a Memory Map for Software Development

When connected to the Nios II processor, SOPC Builder generates a header file that defines the address of each Avalon-MM slave component. In addition, each slave component can provide software drivers and other software functions and libraries for the processor.

How you write software for the system depends heavily on the nature of the processor in the system. For example, Nios II processor systems use Nios II processor-specific software development tools. These tools are separate from SOPC Builder, but they do use the output of SOPC Builder as the foundation for software development.

Creating a Simulation Model and Test Bench

You can simulate your custom systems with minimal effort immediately after generating the system with SOPC Builder. During system generation, SOPC Builder optionally outputs a push-button simulation environment that eases the system simulation effort. SOPC Builder generates both a simulation model and a testbench for the entire system. The testbench includes the following functionality:

- Instantiates the system module
- Drives all clocks and resets appropriately
- Optionally instantiates simulation models for off-chip devices

Getting Started

One of the easiest ways to get started using SOPC Builder is to read the *Nios II Hardware Development Tutorial* which guides you step by step in building a microprocessor system, including CPU, memory, and peripherals. This tutorial and other SOPC Builder example designs are included in the Nios II Embedded Design Suite (EDS). You can download this design suite for free from the Altera Download Center at **www.altera.com/download**.

Referenced Documents

This chapter references the following documents:

- *Avalon Memory-Mapped Interface Specification*
- *System Interconnect Fabric for Streaming Interfaces*
- *Avalon Streaming Interface Specification*
- *SOPC Builder Components*
- *Component Editor*
- *System Interconnect Fabric for Memory-Mapped Interfaces*
- *Nios II Hardware Development Tutorial*

Document Revision History

Table 1–1 shows the revision history for this chapter.

Table 1–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007, v7.2.0	• Updated with new 7.2 functionality and terminology. Deleted unneeded description of SOPC Builder Ready Components.	—
May 2007, v7.1.0	• Updated Avalon terminology because of changes to Avalon technologies. Changed old "Avalon switch fabric" term to "system interconnect fabric." Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface." • Added new information on Avalon Streaming (Avalon-ST) interface. • Revised system module block diagram • Added Referenced Documents section.	This chapter was revised to introduce the Avalon streaming interface in addition to the Avalon Memory-Mapped interface. The block diagram was made more comprehensive.
March 2007, v7.0.0	No change from previous release	—
November 2007, v6.1.0	No change from previous release.	—
May 2006, v6.0.0	No change from previous release.	—
October 2005, v5.1.0	No change from previous release.	—
May 2005, v5.0.0	No change from previous release.	—
February 2005, v1.0	Initial release.	—

2. System Interconnect Fabric for Memory-Mapped Interfaces

QII54003-7.2.0

Introduction

System interconnect fabric for memory-mapped interfaces is a high-bandwidth interconnect structure for connecting components that use the Avalon® Memory-Mapped (Avalon-MM) interface. System interconnect fabric consumes minimal logic resources and provides greater flexibility than a typical shared system bus. This is a cross-connect fabric and not a tristated or time-sliced shared medium.This chapter describes the functions of system interconnect fabric for memory-mapped interfaces and the implementation of those functions.

High-Level Description

System interconnect fabric is the collection of interconnect and logic resources that connects Avalon-MM master and slave ports on components in a system. SOPC Builder generates system interconnect fabric to match the needs of the specific components in a system. System interconnect fabric encapsulates the connection details of a system. It guarantees that signals travel correctly between master and slave ports, as long as the ports adhere to the rules of the Avalon Memory-Mapped interface specification. This chapter provides information on the following topics:

- "Address Decoding" on page 2–5
- "Datapath Multiplexing" on page 2–6
- "Wait-State Insertion" on page 2–7
- "Pipeline Read Transfers" on page 2–8
- "Native Address Alignment and Dynamic Bus Sizing" on page 2–9
- "Arbitration for Multimaster Systems" on page 2–12
- "Burst Management" on page 2–18
- "Clock Domain Crossing" on page 2–19
- "Interrupts" on page 2–29
- "Reset Distribution" on page 2–31



For details about the Avalon-MM interface, refer to the *Avalon Memory-Mapped Interface Specification*

System interconnect fabric for memory-mapped interfaces supports:

- Any number of master and slave components. The master-to-slave relationship can be one-to-one, one-to-many, many-to-one, or many-to-many.
- On-chip components

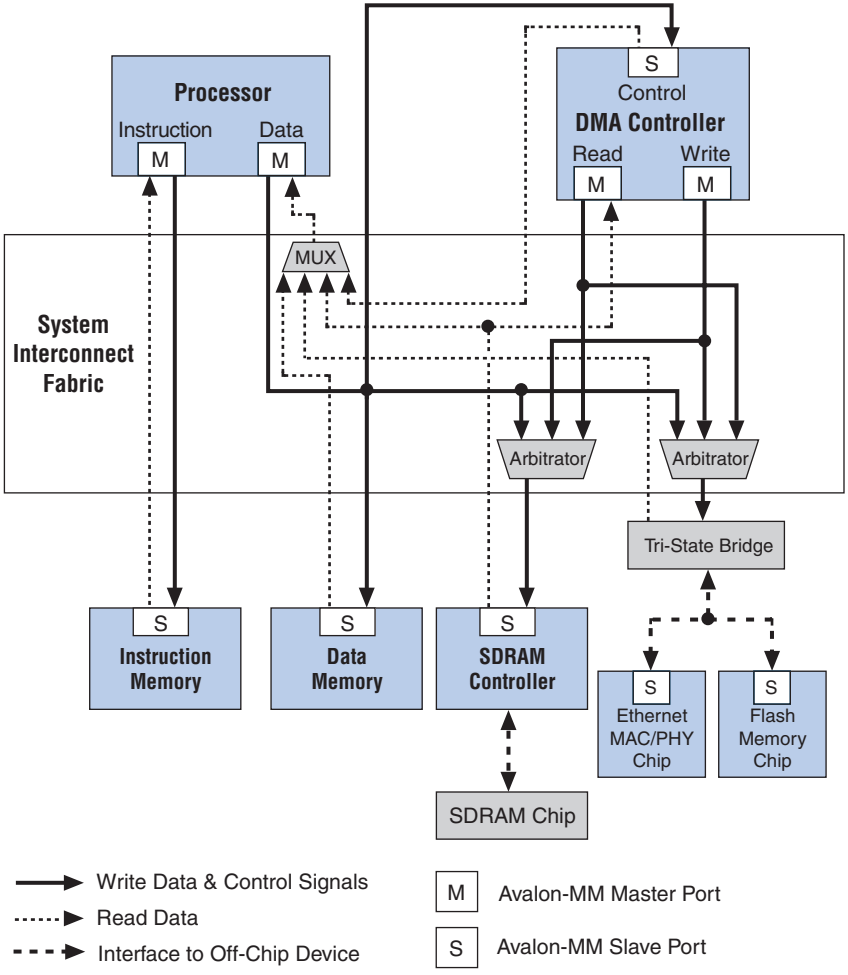
- Interfaces to off-chip devices
- Master and slave ports of differing data widths
- Big-endian or little-endian components
- Components operating in different clock domains
- Components using multiple Avalon-MM ports

Figure 2–1 shows a simplified diagram of the system interconnect fabric in an example memory-mapped system with multiple masters.



All figures in this chapter are simplified to show only the particular function being discussed. In a complete system, the system interconnect fabric might alter the address, data, and control paths beyond what is shown in any one particular figure.

Figure 2–1. System Interconnect Fabric—Example System



SOPC Builder supports components with multiple Avalon-MM ports, such as the processor component shown in Figure 2–1. Because SOPC Builder can create system interconnect fabric to connect components with multiple ports, you can create complex interfaces that provide more functionality than a single Avalon-MM port. For example, you can create a component with two different Avalon-MM slaves, each with an associated interrupt interface.

System interconnect fabric can connect any topology of component connections, as long as each port conforms to the Avalon interface specification. It can, for example, connect a system comprised of only two components with unidirectional dataflow between them. Avalon-MM interfaces are suitable for random addressable transactions, such as to memories or embedded peripherals. Avalon-ST interfaces are suitable for dataflow interconnection, as found in packet processing or DSP pipelines.



For more information, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook* and the *Avalon Streaming Interface Specification*.

Generating system interconnect fabric is SOPC Builder's primary purpose. SOPC Builder can be used to manage and edit your design. Because SOPC Builder automatically generates system interconnect fabric, you may not be required to interact directly with it or the HDL that describes it; however, a basic understanding of how it works can help you optimize your components and systems. For example, knowledge of the arbitration mechanism can help designers of multimaster systems minimize the impact of arbitration on the system throughput.

Fundamentals of Implementation

System interconnect fabric for memory-mapped interfaces implements a switched interconnect structure that provides concurrent paths between master and slave ports. System interconnect fabric consists of synchronous logic and routing resources inside the FPGA.

For each port interface on components, system interconnect fabric manages Avalon-MM transfers, interacting with and responding to signals on the connected component. The signals that appear on the master port and corresponding slave port of a master-slave pair can be different. In the path between master and slave ports, the system interconnect fabric might introduce registers for timing synchronization, finite state machines for event sequencing, or nothing at all, depending on the services required by the specific ports.

Functions of System Interconnect Fabric

System interconnect fabric logic provides the following functions:

- "Address Decoding" on page 2–5
- "Datapath Multiplexing" on page 2–6
- "Wait-State Insertion" on page 2–7
- "Pipeline Read Transfers" on page 2–8
- "Native Address Alignment and Dynamic Bus Sizing" on page 2–9
- "Arbitration for Multimaster Systems" on page 2–12

- "Burst Management" on page 2–18
- "Clock Domain Crossing" on page 2–19
- "Interrupts" on page 2–29
- "Reset Distribution" on page 2–31

The behavior of these functions in a specific SOPC Builder system depends on the design of the components in the system and the settings made in SOPC Builder. The remaining sections of this chapter describe how SOPC Builder implements each function.

Address Decoding

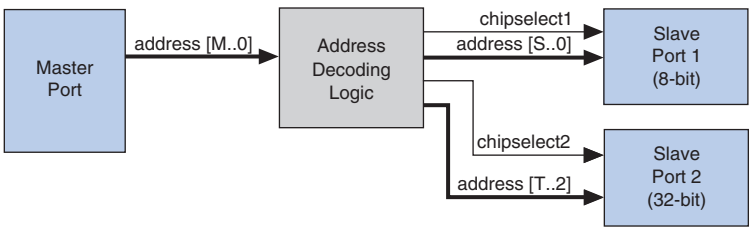
Address decoding logic in the system interconnect fabric distributes an appropriate address and produces a `chipselect` signal for each slave port. Address decoding logic simplifies component design in the following ways:

- The system interconnect fabric selects a slave port whenever it is being addressed by a master. Slave components do not need to decode the address to determine when they are selected.
- Slave port addresses are properly aligned for the slave port.
- SOPC Builder automatically generates address decoding logic to implement the memory map specified in the GUI. Therefore, changing the system memory map does not involve manually editing HDL.

Figure 2–2 shows a block diagram of the address-decoding logic for one master and two slave ports. Separate address-decoding logic is generated for every master port in a system.

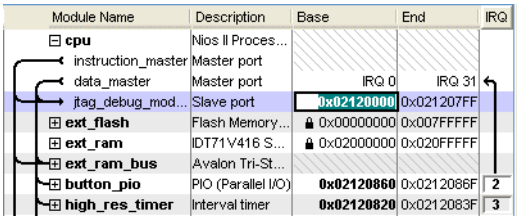
As shown in Figure 2–2, the address decoding logic handles the difference between the master address width (*M*) and the individual slave address widths (*S* and *T*). It also maps only the necessary master address bits to access words in each slave port's address space.

Figure 2–2. Block Diagram of Address Decoding Logic



In SOPC Builder, the user-configurable aspects of address decoding logic are controlled by the **Base** setting in the list of active components on the **System Contents** tab, as shown in Figure 2–3.

Figure 2–3. Base Settings in SOPC Builder Control Address Decoding

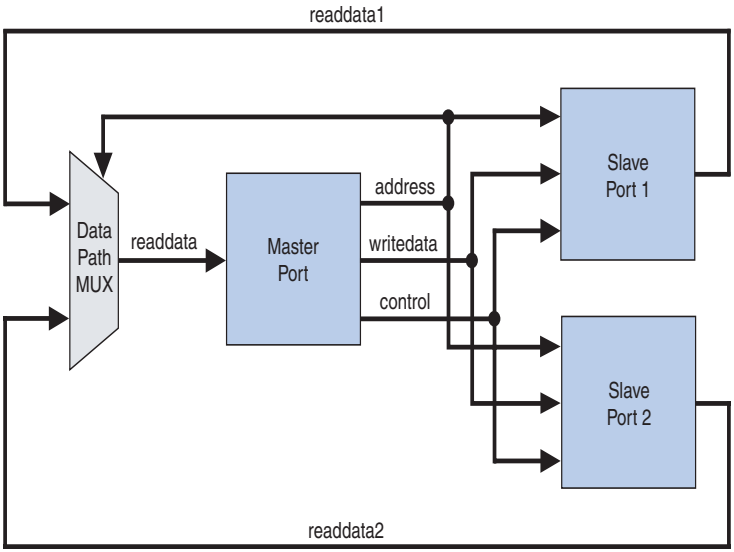


Datapath Multiplexing

Datapath multiplexing logic in the system interconnect fabric drives the writedata from the granted master to the selected slave, and from the readdata from the selected slave back to the requesting master.

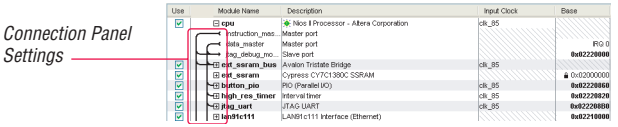
Figure 2–4 shows a block diagram of the datapath multiplexing logic for one master and two slave ports. SOPC Builder generates separate datapath multiplexing logic for every master port in the system.

Figure 2–4. Block Diagram of Datapath Multiplexing Logic



In SOPC Builder, the generation of datapath multiplexing logic is specified using the connections panel on the **System Contents** page, as shown in Figure 2–5.

Figure 2–5. Connection Panel Settings in SOPC Builder Control Datapath Multiplexing

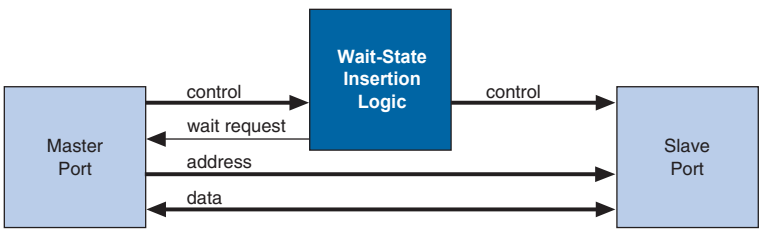


Wait-State Insertion

Wait states extend the duration of a transfer by one or more cycles. Wait-state insertion logic accommodates the timing needs of each slave port, and coordinates the master port to wait until the slave can proceed. System interconnect fabric inserts wait states into a transfer when the target slave port cannot respond in a single clock cycle. System interconnect fabric also inserts wait states in cases when slave read-enable and write-enable signals have setup or hold time requirements.

Wait-state insertion logic is a small finite-state machine that translates control signal sequencing between the slave side and the master side. Figure 2–6 shows a block diagram of the wait-state insertion logic between one master and one slave.

Figure 2–6. Block Diagram of Wait-State Insertion Logic



System interconnect fabric can force a master port to wait for several reasons in addition to the wait state needs of a slave port. For example, arbitration logic in a multimaster system can force a master port to wait until it is granted access to a slave port.

SOPC Builder generates wait-state insertion logic based on the properties of all slave ports in the system.

Pipeline Read Transfers

The Avalon-MM interface supports pipelined read transfers, allowing a pipelined master port to start multiple read transfers in succession without waiting for the prior transfers to complete. Pipelined transfers allow master-slave pairs to achieve higher throughput, even though the slave port might require one or more cycles of latency to return data for each transfer.

SOPC Builder generates system interconnect fabric with pipeline management logic to take advantage of pipelined components wherever possible, based on the pipeline properties of each master-slave pair in the system. Regardless of the pipeline latency of a target slave port, SOPC Builder guarantees that read data arrives at each master port in the order requested. Because master and slave ports often have mismatched pipeline latency, system interconnect fabric often contains logic to reconcile the differences. Many cases are possible, as shown in Table 2–1.

Table 2–1. Various Cases of Pipeline Latency in a Master-Slave Pair

Master Port	Slave Port	Pipeline Management Logic Structure
No Pipeline	No Pipeline	The system interconnect fabric does not instantiate logic to handle pipeline latency.
No Pipeline	Pipelined with Fixed or Variable Latency	The system interconnect fabric forces the master port to wait through any slave-side latency cycles. This master-slave pair gains no benefits of pipelining, because the master port is not pipelined and therefore waits for each transfer to complete before beginning a new transfer. However, while the master port is waiting, the slave port can accept transfers from a different master port.
Pipelined	No Pipeline	The system interconnect fabric carries out the transfer as if neither port were pipelined, forcing the master port to wait until the slave port returns data.
Pipelined	Pipelined with Fixed Latency	The system interconnect fabric coordinates the master port to capture data at the exact clock cycle when data is valid on the slave port. This case enables this master-slave pair to achieve maximum throughput performance.
Pipelined	Pipelined with Variable Latency	This is the simplest pipelined case, in which the slave port asserts a signal when its readdata is valid, and the master port captures the data. This case enables this master-slave pair to achieve maximum throughput performance.

SOPC Builder generates logic to handle pipeline latency based on the properties of the master and slave ports in the system. When configuring a system in SOPC Builder, there are no settings that directly control the pipeline management logic in the system interconnect fabric.

Native Address Alignment and Dynamic Bus Sizing

SOPC Builder generates system interconnect fabric to accommodate master and slave ports with unmatched data widths. Address alignment affects how slave data is aligned in a master port's address space, in the case that the master and slave data widths are different. Address alignment is a property of each slave port, and can be different for each slave port in a system. A slave port can declare itself to use one of the following:

- Native address alignment
- Dynamic bus sizing

Table 2–2 demonstrates native address alignment and dynamic bus sizing for a 32-bit master port connected to a 16-bit slave port (a 2:1 ratio). In this example, the slave port is mapped to base address *BASE* in the master port's address space. In Table 2–2, `OFFSET` refers to the offset into the 16-bit slave address space.

Table 2–2. 32-Bit Master View of 16-Bit Slave Data

32-bit Master Address	Data with Native Alignment	Data with Dynamic Bus Sizing
BASE + 0x0 (word 0)	`0x0000:OFFSET[0]`	`OFFSET[1]:OFFSET[0]`
BASE + 0x4 (word 1)	`0x0000:OFFSET[1]`	`OFFSET[3]:OFFSET[2]`
BASE + 0x8 (word 2)	`0x0000:OFFSET[2]`	`OFFSET[5]:OFFSET[4]`
BASE + 0xC (word 3)	`0x0000:OFFSET[3]`	`OFFSET[7]:OFFSET[6]`
...	. . .	. . .
BASE + 4*N* (word *N*)	`0x0000:OFFSET[N]`	`OFFSET[2N+1]:OFFSET[2N]`

SOPC Builder generates appropriate address-alignment logic based on the properties of the master and slave ports in the system. When configuring a system in SOPC Builder, there are no settings that directly control the address alignment in the system interconnect fabric.

Dynamic Bus Sizing

Dynamic bus sizing hides the details of interfacing a narrow component device to a wider master port, and vice versa. When an *N*-bit master port accesses a slave port with dynamic bus sizing, the master port operates exclusively on full *N*-bit words of data, without awareness of the slave data width.



When using dynamic bus sizing, the slave data width with units of bytes must be a power of two.

Dynamic bus sizing provides the following benefits:

- Eliminates the need to create address-alignment hardware manually.
- Reduces design complexity of the master component.
- Enables any master port to access any memory device, regardless of the data width.

In the case of dynamic bus sizing, the system interconnect fabric includes a small finite state machine that reconciles the difference between master and slave data widths. The behavior is different depending on whether the master data width is wider or narrower than the slave.

Wider Master

In the case of a wider master, the dynamic bus-sizing logic accepts a single, wide transfer on the master side, and then performs multiple narrow transfers on the slave side. For a data-width ratio of *N*:1, the dynamic bus-sizing logic generates up to *N* slave transfers for each master transfer. The master port waits while multiple slave-side transfers complete; the master transfer ends when all slave-side transfers end.

Dynamic bus-sizing logic uses the master-side byte-enable signals to generate appropriate slave transfers. The dynamic bus-sizing logic performs as many slave-side transfers as necessary to write or read the specified byte lanes.

Narrower Master

In the case of a narrower master, one transfer on the master side generates one transfer on the slave side. In this case, multiple master word addresses map to a single offset in the slave memory space. The dynamic bus-sizing logic maps each master address to a subset of byte lanes in the appropriate slave offset. All bytes of the slave memory are accessible in the master address space.

Table 2–3 demonstrates the case of a 32-bit master port accessing a 64-bit slave port with dynamic bus sizing. In the table, offset refers to the offset into the slave port memory space.

Table 2–3. 32-Bit Master View of 64-Bit Slave with Dynamic Bus Sizing (Part 1 of 2)

32-bit Address	Data
0×00000000 (word 0)	OFFSET[0]$_{31..0}$
0×00000004 (word 1)	OFFSET[0]$_{63..32}$

Table 2–3. 32-Bit Master View of 64-Bit Slave with Dynamic Bus Sizing (Part 2 of 2)

32-bit Address	Data
0×00000008 (word 2)	OFFSET[1]$_{31..0}$
0×0000000C (word 3)	OFFSET[1]$_{63..32}$

In the case of a read transfer, the dynamic bus-sizing logic multiplexes the appropriate byte lanes of the slave data to the narrow master port. In the case of a write transfer, the dynamic bus-sizing logic uses slave-side byte-enable signals to write only to the appropriate byte lanes.

Native Address Alignment

Slave ports that access address-mapped registers inside the component generally use native address alignment. The defining properties of native address alignment are:

- Each slave offset (that is, word) maps to exactly one master word, regardless of the data width of the ports.
- One transfer on the master port generates exactly one transfer on the slave port.

In the case of native address alignment, system interconnect fabric maps all slave data bits to the lower bits of the master data, and fills any remaining upper bits with zero. System interconnect fabric performs simple wire-mapping in the datapath, but nothing else.

Native address alignment is only valid if the master data width is equal to or wider than the slave data width. If an *N*-bit master port is connected to a wider slave with native alignment, then the master port can access only the lower *N* data bits at each offset in the slave.



Native address alignment prevents use of the slave with narrow masters and some bridge implementations, and is not recommended for new components.

Arbitration for Multimaster Systems

System interconnect fabric supports systems with multiple master components. In a system with multiple master ports, such as the system pictured in Figure 2–1 on page 2–3, the system interconnect fabric provides shared access to slave ports using a technique called slave-side arbitration. Slave-side arbitration determines which master port gains access to a specific slave port in the event that multiple master ports attempt to access the same slave port at the same time.

The multimaster architecture used by system interconnect fabric offers the following benefits:

- Eliminates the need to create arbitration hardware manually.
- Allows multiple master ports to transfer data simultaneously. Unlike traditional host-side arbitration architectures in which each master must wait until it is granted access to the shared bus, multiple Avalon-MM masters can simultaneously perform transfers with independent slaves. Arbitration logic stalls a master port only when multiple master ports attempt to access the same slave port during the same cycle.
- Eliminates unnecessary master-slave connections. The connection between a master port and a slave port exists only if it is specified in SOPC Builder. If a master port never initiates transfers to a specific slave port, no connection is necessary, and therefore SOPC Builder does not waste logic resources to connect the two ports.
- Provides configurable arbitration settings, and arbitration for each slave port is specified independently. For example, you can grant one master port the most access to a particular slave port, while other master ports have more access to other slave ports.
- Simplifies master component design. The details of arbitration are encapsulated inside the system interconnect fabric. Each Avalon-MM master port connects to the system interconnect fabric as if it is the only master port in the system. As a result, you can reuse a component in single-master and multimaster systems without requiring design changes to the component.

This section discusses the architecture of the system interconnect fabric generated by SOPC Builder for multimaster systems.

Traditional Shared Bus Architectures

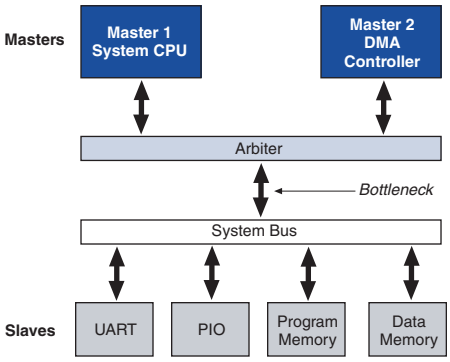
As a frame of reference for the discussion of multiple masters and arbitration, this section describes traditional bus architectures.

In traditional bus architectures, one or more bus masters and bus slaves connect to a shared bus, consisting of wires on a printed circuit board. A single arbiter controls the bus (that is, the path between bus masters and bus slaves), so that multiple bus masters do not simultaneously drive the

bus. Each bus master requests control of the bus from the arbiter, and the arbiter grants access to a single master at a time. Once a master has control of the bus, the master performs transfers with a bus slave. If multiple masters attempt to access the bus at the same time, the arbiter allocates the bus resources to a single master based on fixed arbitration rules, forcing all other masters to wait. For example, the priority arbitration scheme—in which the arbiter always grants control to the master with the highest priority—is used in many existing bus architectures.

Figure 2–7 illustrates the bus architecture for a traditional processor system. Access to the shared system bus becomes the bottleneck for throughput: only one master has access to the bus at a time, which means that other masters are forced to wait and only one slave can transfer data at a time.

Figure 2–7. Bus Architecture in a Traditional Microprocessor System



Slave-Side Arbitration

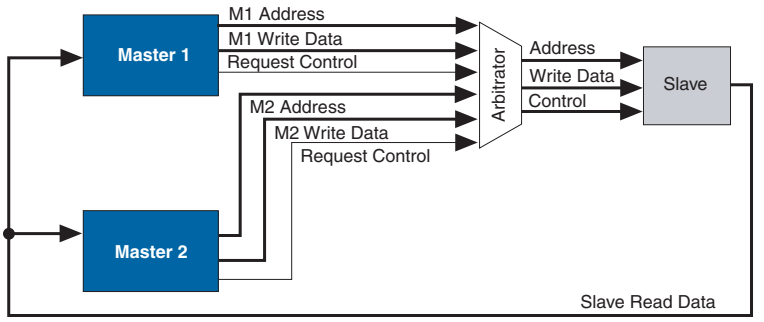
The multimaster architecture used by system interconnect fabric eliminates the bottleneck for access to a shared bus, because the system does not have shared bus signals. Avalon-MM master-slave pairs are connected by dedicated paths. A master port never waits to access a slave port, unless a different master port attempts to access the same slave port at the same time. As a result, multiple master ports can be active at the same time, simultaneously transferring data with independent slave ports.

A multimaster Avalon-MM system requires arbitration, but only when two masters contend for the same slave port. This arbitration is called slave-side arbitration, because it is implemented at the point where two (or more) master ports connect to a single slave. Master ports contend for individual slave ports, not for a shared bus resource.

For example, Figure 2–1 on page 2–3 demonstrates a system with two master ports (a CPU and a DMA controller) sharing a slave port (an SDRAM controller). Arbitration is performed at the SDRAM slave port; the arbiter dictates which master port gains access to the slave port if both master ports initiate a transfer with the slave port in the same cycle.

Figure 2–8 focuses on the two master ports and the shared slave port, and shows additional detail of the data, address, and control paths. The arbiter logic multiplexes all address, data, and control signals from a master port to a shared slave port.

Figure 2–8. Detailed View of Multimaster Connections



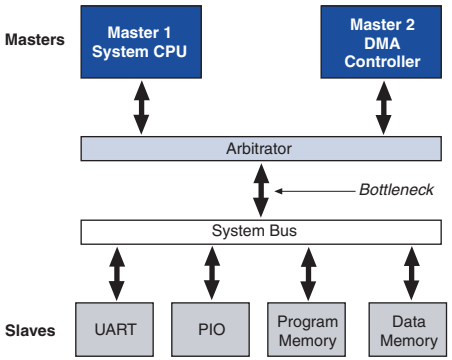
Arbiter Details

SOPC Builder generates an arbiter for every slave port, based on arbitration parameters specified in SOPC Builder. The arbiter logic performs the following functions for its slave port:

- Evaluates the address and control signals from each master port and determines which master port, if any, gains access to the slave next.
- Grants access to the chosen master port and forces all other requesting master ports to wait.
- Uses multiplexers to connect address, control, and datapaths between the multiple master ports and the slave port.

Figure 2–9 shows the arbiter logic in an example multimaster system with two master ports, each connected to two slave ports.

Figure 2–9. Block Diagram of Arbiter Logic



Arbitration Rules

This section describes the rules by which the arbiter grants access to master ports when they contend.

Setting Arbitration Parameters in SOPC Builder

You specify the arbitration shares for each master using the connection panel on the **System Contents** tab of SOPC Builder, as shown in Figure 2–10.

Figure 2–10. Arbitration Settings on the System Contents Tab

Module Name	Description	Clock
⊟ **cpu**	Nios II Processor - Alte...	clk
instruction_master	Master port	
data_master	Master port	
1 1 → jtag_debug_module	Slave port	
1 ⊞ **sys_clk_timer**	Interval timer	clk
1 1 ⊞ **ext_ram_bus**	Avalon Tri-State Bridge	clk
⊞ **ext_flash**	Flash Memory (Commo...	
⊞ **ext_ram**	IDT71V416 SRAM	
1 1 ⊞ **epcs_controller**	EPCS Serial Flash Cont...	clk
⊞ **lan91c111**	LAN91c111 Interface (...	
1 ⊞ **jtag_uart**	JTAG UART	clk

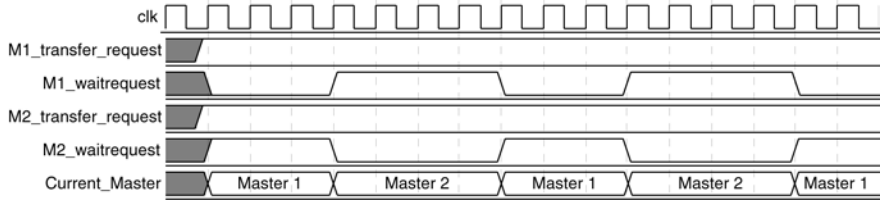
The arbitration settings are hidden by default. To see them, on the View menu, click **Show Arbitration**.

Fairness-Based Shares

Arbiter logic uses a fairness-based arbitration scheme. In a fairness-based arbitration scheme, each master port pair has an integer value of *transfer shares* with respect to a slave port. One share represents permission to perform one transfer.

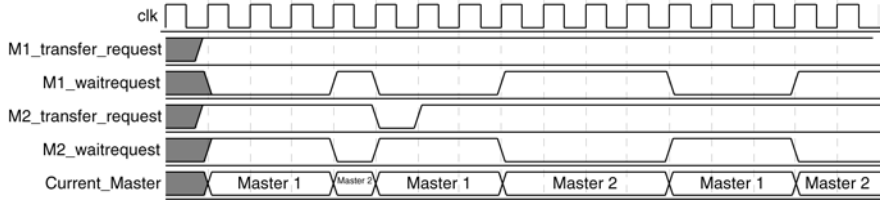
For example, assume that two master ports continuously attempt to perform back-to-back transfers to a slave port. Master 1 is assigned three shares and Master 2 is assigned four shares. In this case, the arbiter grants Master 1 access for three transfers, then Master 2 for four transfers. This cycle repeats indefinitely. Figure 2–11 demonstrates this case, showing each master port's transfer request output, wait request input (which is driven by the arbiter logic), and the current master with control of the slave.

Figure 2–11. Arbitration of Continuous Transfer Requests from Two Master Ports



If a master stops requesting transfers before it exhausts its shares, it forfeits all its remaining shares, and the arbiter grants access to another requesting master. See Figure 2–12. After completing one transfer, Master 2 stops requesting for one clock cycle. As a result, the arbiter grants access back to Master 1, which gets a replenished supply of shares.

Figure 2–12. Arbitration of Two Masters with a Gap in Transfer Requests



Round-Robin Scheduling

When multiple master ports contend for access to a slave port, the arbiter grants shares in round-robin manner. Round-robin scheduling drives a request interface according to space available and data available credit interfaces. At every slave transfer, only requesting master ports are included in the arbitration.

Burst Transfers

Avalon-MM burst transfers grant a master port uninterrupted access to a slave port for a specified number of transfers. The master port specifies the number of transfers when it initiates the burst. Once a burst begins between a master-slave pair, arbiter logic does not allow any other master port to access the slave port until the burst completes. For further information, refer to "Burst Management" on page 2–18.

Minimum Share Value

A component design can declare the minimum number of shares in each round-robin cycle, which affects how the arbiter grants access. For example, if a slave port has a minimum share value of ten, then the arbiter will grant at least ten shares to any master port when it begins a sequence of transfer requests. The arbiter might grant more shares, if the master port is assigned more shares in SOPC Builder.

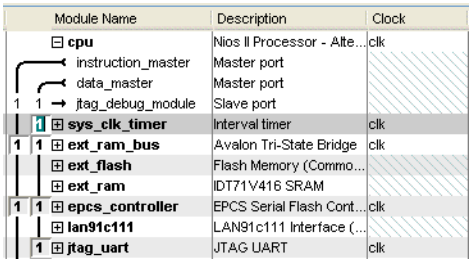
By declaring a minimum share value of *N*, a slave port declares that it is more efficient at handling continuous sequential transfers of length *N*. Accessing the slave port in sequences less than *N* incurs performance penalties that might prevent the slave port from achieving higher performance. By nature, continuous back-to-back master transfers tend to access sequential addresses. However, there is no requirement that the master port perform transfers to sequential addresses.



Burst transfers provide even higher performance for continuous transfers when they are guaranteed to access sequential addresses. The minimum share value does not apply to slave ports that support bursts; the burst length takes precedence over minimum share value. Refer to "Burst Management" on page 2–18 for information.

You specify the arbitration shares for each master using the connection panel on the **System Contents** tab of SOPC Builder, as shown in Figure 2–13.

Figure 2–13. Arbitration Settings on the System Contents Tab



The arbitration settings are hidden by default. To see them, on the View menu, click **Show Arbitration**.

Burst Management

System interconnect fabric provides burst management logic to accommodate the burst capabilities of each port in the system, including ports that do not support burst transfers. Burst management logic is a finite state machine that translates the sequencing of address and control signals between the slave side and the master side.

The maximum burst length for each port is determined by the component design and is independent of other ports in the system. Therefore, a particular master port might be capable of initiating a burst longer than a slave port's maximum supported burst length. In this case, the burst management logic translates the master burst into smaller slave bursts, or into individual slave transfers if the slave port does not support bursts. Until the master port completes the burst, the arbiter logic prevents other master ports from accessing the target slave port.

For example, if a master port initiates a burst of 16 transfers to a slave port with maximum burst length of 8, the burst management logic initiates two bursts of length 8 to the slave port. If a master port initiates a burst of 16 transfers to a slave port that does not support bursts, the burst management logic initiates 16 separate transfers to the slave port.

Clock Domain Crossing

SOPC Builder generates clock-domain crossing (CDC) logic that hides the details of interfacing components operating in asynchronous clock domains. The system interconnect fabric upholds the Avalon-MM protocol with each port independently, and therefore each Avalon-MM port need only be aware of its own clock domain. The system interconnect fabric logic propagates transfers across clock domain boundaries automatically.

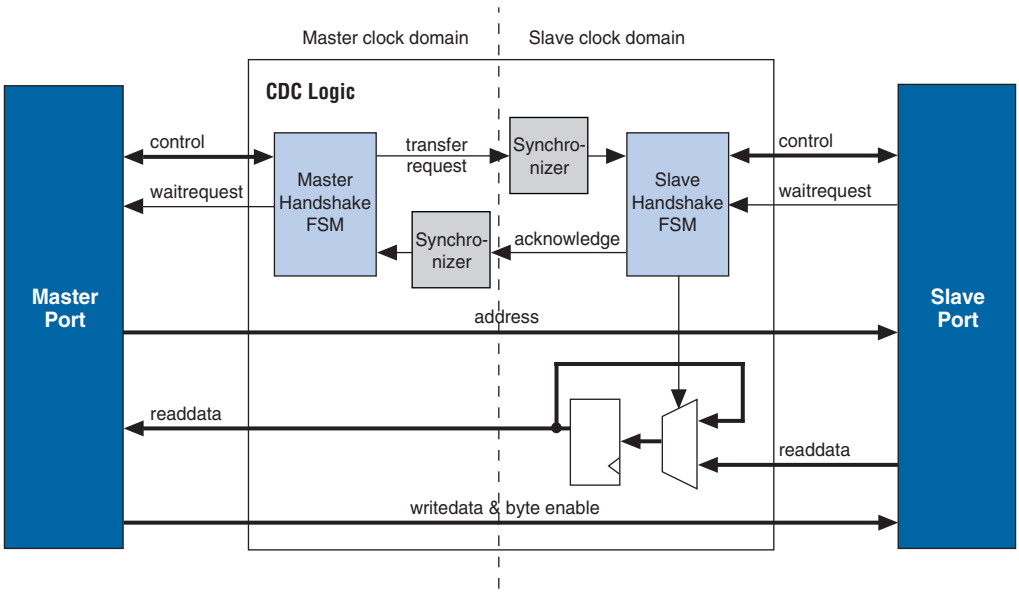
The CDC logic in system interconnect fabric provides the following benefits that simplify system design efforts:

- Allows component interfaces to operate at a different clock frequency than system logic.
- Eliminates the need to design CDC hardware manually.
- Each Avalon-MM port operates in only one clock domain, which reduces design complexity of components.
- Enables master ports to access any slave port without communication with the slave clock domain.
- Allows you to focus performance optimization efforts only on components that require fast clock speed.

Description of Clock Domain-Crossing Logic

The CDC logic consists of two finite state machines (FSM), one in each clock domain, that use a simple hand-shaking protocol to propagate transfer control signals (read request, write request, and the master wait-request signals) across the clock boundary. Figure 2–14 shows a block diagram of the clock domain crossing logic between one master and one slave port.

Figure 2–14. Block Diagram of Clock Domain-Crossing Logic



The Synchronizer blocks in Figure 2–14 use multiple stages of flip-flops to eliminate the propagation of metastable events on the control signals that enter the handshake FSMs.

The CDC logic works with any clock ratio. Altera® tests the CDC logic extensively on a variety of system architectures, both in simulation and in hardware, to ensure that the logic functions correctly.

The typical sequence of events for a transfer across the CDC logic is described below:

1. Master port asserts address, data, and control signals.
2. The master handshake FSM captures the control signals, and immediately forces the master port to wait.



The FSM uses only the control signals, not address and data. For example, the master port simply holds the address signal constant until the slave side has safely captured it.

3. Master handshake FSM initiates a transfer request to the slave handshake FSM.
4. The transfer request is synchronized to the slave clock domain.
5. The slave handshake FSM processes the request, performing the requested transfer with the slave port.
6. When the slave transfer completes, the slave handshake FSM sends an acknowledge back to the master handshake FSM.
7. The acknowledge is synchronized back to the master clock domain.
8. The master handshake FSM completes the transaction by releasing the master port from the wait condition.

Transfers proceed as normal on the slave and the master side, without a special protocol to handle crossing clock domains. From the perspective of a slave port, there is nothing different about a transfer initiated by a master port in a different clock domain. From the perspective of a master port, a transfer across clock domains simply requires extra clock cycles. Similar to other transfer delay cases (for example, arbitration delay or wait states on the slave side), the system interconnect fabric simply forces the master port to wait until the transfer terminates. As a result, latency-aware master ports do not benefit from pipelining when performing transfers to a different clock domain.

Location of Clock Domain Crossing Logic

SOPC Builder automatically determines where to insert the CDC logic, based on the system contents and the connections between components. SOPC Builder places CDC logic to maintain the highest transfer rate for all components. SOPC Builder evaluates the need for CDC logic on each slave port independently, and generates CDC logic wherever necessary.

Duration of Transfers Crossing Clock Domains

CDC logic extends the duration of master transfers across clock domain boundaries. In the worst case, each transfer is extended by five master clock cycles and five slave clock cycles. The components of this delay are the following:

- Four additional master clock cycles, due to the master-side clock synchronizer
- Four additional slave clock cycles, due to the slave-side clock synchronizer
- One additional clock in each direction, due to potential metastable events as the control signals cross clock domains

Systems that require higher performance clock crossing logic should use the Avalon-MM clock crossing bridge instead of the automatically inserted CDC logic. The clock-crossing bridge includes a buffering mechanism, so that multiple reads and writes can be pipelined. After paying the initial penalty for the first read or write, there is no additional latency penalty for pending reads and writes, increasing throughput by up to four times, at the expense of added logic resources.

For more information, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*.

Implementing Multiple Clock Domains in SOPC Builder

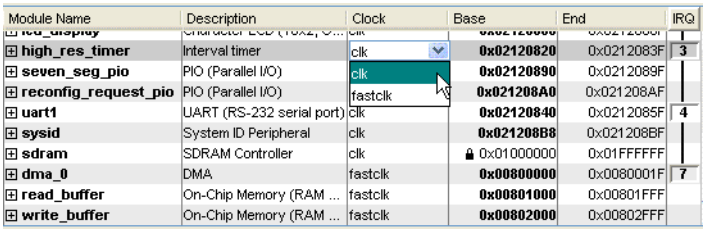
You specify the clock domains used by your system on the **System Contents** tab of SOPC Builder. You define the input clocks to the system with the **Clock Settings** table, shown in Figure 2–15. Clock sources can be driven by external input signals to the system module, or by PLLs inside the system module. Clock domains are differentiated based on the name of the clock. It is possible to create multiple asynchronous clocks with the same frequency.

Figure 2–15. Clock Settings on the System Contents Tab

Clock	Source	MHz	Pipeline
clk_85	External	85.0	☐
clk_233	c0 from pll	233.75	☑
click to add...			☐

You specify which clock drives which components using the table of active components after you define the system clocks, as shown in Figure 2–16.

Figure 2–16. Assigning Clocks to Components



Alternatively, the clock patch panel can be used.

This section describes the hardware structure and functionality of the Avalon-MM clock-crossing bridge component.

Component Overview

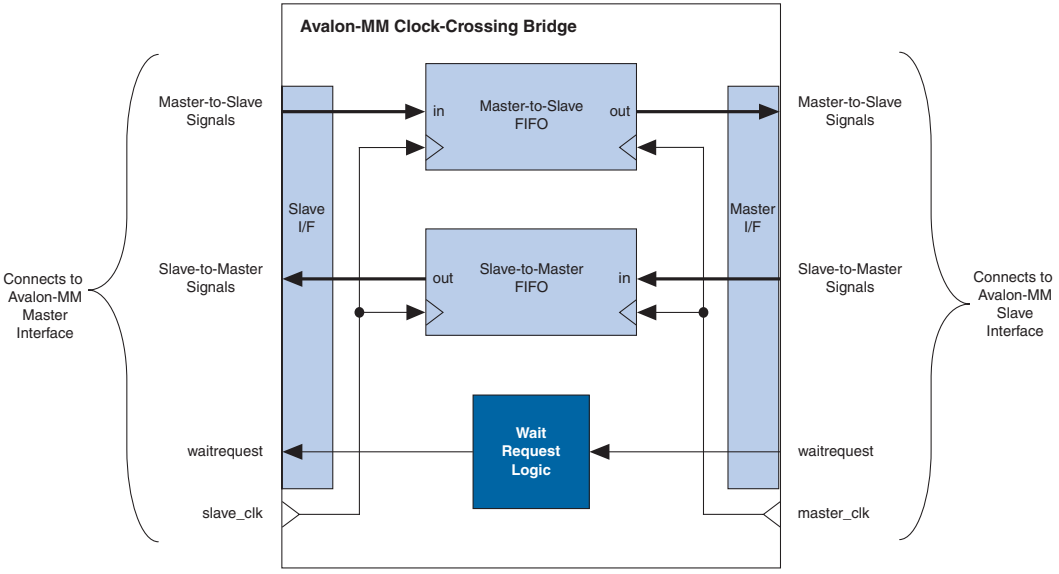
The Avalon-MM clock-crossing bridge allows you to connect Avalon-MM master and slave ports that operate in different clock domains. Without a bridge, SOPC Builder automatically includes generic CDC logic in the system interconnect fabric, but it does not provide optimal performance for high-throughput applications. The CDC logic uses a four-way handshake mechanism so that each read and write takes multiple cycles in each direction. Because the clock-crossing bridge includes a buffering mechanism, you can pipeline multiple reads and writes. After an initial penalty for the first read or write, there is no additional latency penalty for pending reads and writes, increasing throughput by up to four times, at the expense of additional logic resources. The clock-crossing bridge has parameterizeable FIFOs for master-to-slave and slave-to-master signals, which allows burst transfers across clock domains.

The Avalon-MM clock-crossing bridge component is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

Functional Description

Figure 2–17 shows a block diagram of the Avalon-MM clock-crossing bridge component. The following sections describe the component's hardware functionality.

Figure 2–17. Avalon-MM Clock-Crossing Bridge Block Diagram



Interfaces

The bridge interface comprises an Avalon-MM slave port and an Avalon-MM master port. The data width of the ports is configurable, which affects the size of the bridge hardware and how SOPC Builder generates dynamic bus sizing logic in the system interconnect fabric. Both ports support Avalon-MM pipelined transfers with variable latency. Both ports optionally support bursts of user-configurable length.

Clock Domain Crossing Logic and FIFOs

Two FIFOs in the bridge transport address, data, and control signals across the clock-domains. One FIFO captures data traveling in the master-to-slave direction, and the other FIFO captures data in the slave-to-master direction. CDC logic surrounding the FIFOs coordinates the details of passing data across the clock-domain boundaries and ensures that the FIFOs do not overflow or underflow.

The signals that pass through the master-to-slave FIFO include:

- writedata
- address
- read
- write
- nativeaddress
- byteenable
- burstcount, when bursts are allowed.

The signals that pass through the slave-to-master FIFO include:

- readdata
- readdatavalid
- endofpacket

The depth of each FIFO is configurable. Because there are more signals traveling in the master-to-slave direction, changing the depth of the master-to-slave FIFO has a greater impact on the memory utilization of the bridge.

For read transfers across the bridge, the FIFOs in both directions incur latency for data to return from the slave. To avoid paying a latency penalty for each transfer, the master can issue multiple reads which are queued in the FIFO. The slave of the bridge asserts readdatavalid when it drives valid data and asserts waitrequest when it is not ready to accept more reads.

For write transfers, the master-to-slave FIFO causes a delay between the master-to-bridge transfers and the corresponding bridge-to-slave transfers. Because Avalon-MM write transfers do not require an acknowledge from the slave, multiple write transfers from master-to-bridge might complete by the time the bridge initiates the corresponding bridge-to-slave transfers.

Burst Support

The bridge can optionally support bursts with configurable maximum burst length. When configured to support bursts, the bridge propagates bursts between master-slave pairs, up to the maximum burst length. Not having burst support is equivalent to a maximum burst length of one. In this case, the system interconnect fabric automatically deconstructs master-to-bridge bursts into a sequence of individual transfers.

When the bridge is configured to support bursts, the slave-to-master FIFO depth must be configured deeply enough to capture all burst read data without overflowing. The master ports connected to the bridge could potentially fill the master-to-slave FIFO with read burst requests; therefore, the minimum slave-to-master FIFO depth is described in the following equation:

(1)

No Bursts:
minimum depth = master-to-slave FIFO depth + max slave latency;

With Bursts:
minimum depth
*= (master-to-slave FIFO depth + max slave latencv) * (max burst size):*

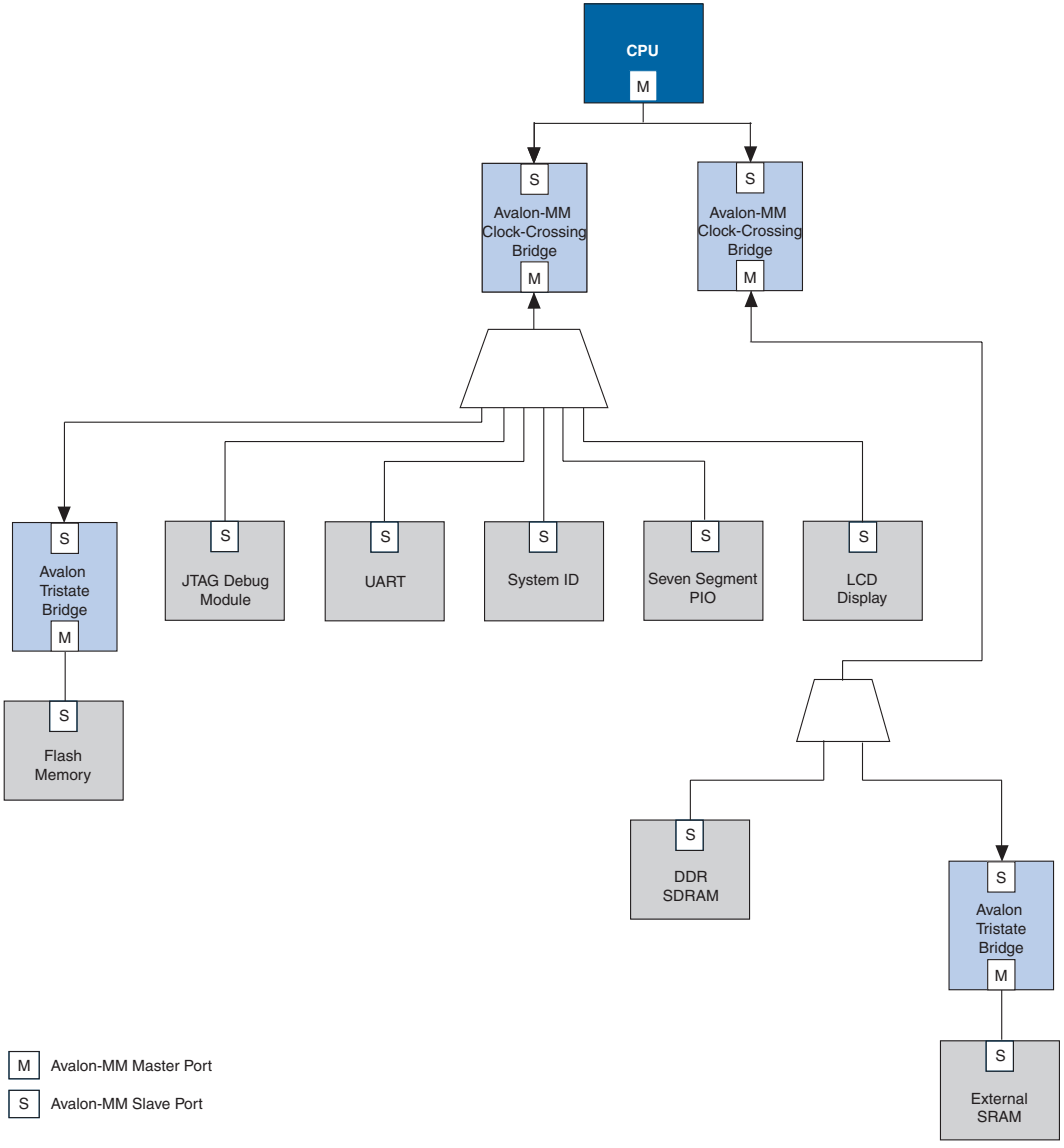


In both cases, the minimum depth is rounded up to the nearest power of two.

Example System with Avalon-MM Clock-Crossing Bridges

Figure 2–18 uses Avalon-MM clocking crossing bridges to separate slave components into two groups. The low-performance slave components are placed behind a single bridge and clocked at a low speed. The high performance components are placed behind a second bridge and clocked at a higher speed. By inserting clock-crossing bridges in the system, you optimize the interconnect fabric and allow the Quartus II Fitter to expend effort optimizing paths that require minimal propagation delay.

Figure 2–18. One Avalon-MM Master with Two Groups of Avalon-MM Slaves



Instantiating the Avalon-MM Clock-Crossing Bridge in SOPC Builder

You use the Avalon-MM Clock-Crossing Bridge MegaWizard® interface in SOPC Builder to specify the hardware features. This section describes the options available on the **Parameter Settings** page of the Megawizard interface.

- **Master-to-Slave FIFO**—These options specify the size and structure of the master-to-slave FIFO.
  - **FIFO Depth**—Determines the depth of the FIFO.
  - **Construct FIFO from registers**—When this option is on, the FIFO uses registers as storage instead of embedded memory blocks. Turning on this option can considerably increase the size of the bridge hardware and lower the f_{MAX}.
- **Slave-to-Master FIFO**—These options specify the size and structure of the slave-to-master FIFO.
  - **FIFO Depth**—Determines the depth of the FIFO.
  - **Construct FIFO from registers**—When this option is on, the FIFO uses registers as storage instead of embedded memory blocks. Turning on this option can considerably increase the size of the bridge hardware.
- **Data Width**—Determines the data width of the master and slave ports on the bridge, and affects the size of both FIFOs.

  For the highest bandwidth, set **Data Width** to be as wide as the widest master port connected to the bridge.
- **Allow Bursts**—Includes logic for the bridge's master and slave ports to support bursts. This option restricts the minimum depth for the slave-to-master FIFO.
- **Maximum Burst Size**—Determines the maximum length of bursts for the bridge to support, when **Allow Bursts** is turned on.

Interrupts

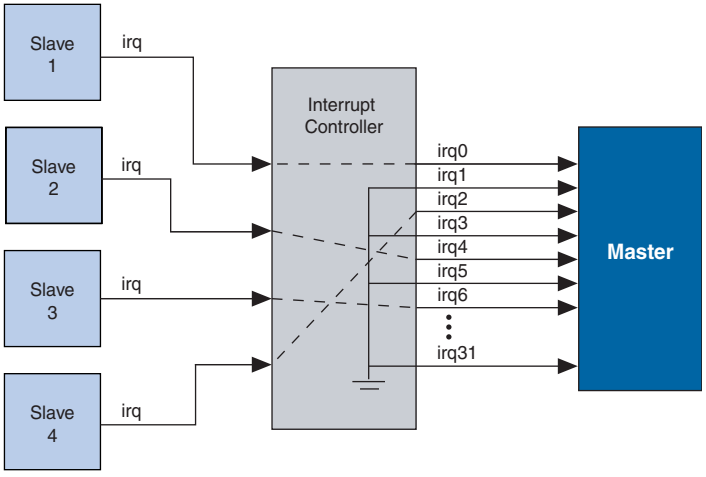
In systems with slave ports that generate interrupt requests (IRQs), the system interconnect fabric includes interrupt controller logic. A separate interrupt controller is generated for each master port that accepts interrupts. The interrupt controller aggregates IRQ signals from all slave ports, and maps slave IRQ outputs to user-specified values on the master IRQ inputs.

Software Priority

In the software priority configuration, the system interconnect fabric passes IRQs directly from slave to master port, without making any assumptions about IRQ priority. In the event that multiple slave ports assert their IRQs simultaneously, the master logic (presumably under software control) determines which IRQ has highest priority, then responds appropriately.

Using software priority, the interrupt controller can handle up to 32 slave IRQ inputs. The interrupt controller generates a 32-bit signal `irq[31..0]` to the master port, and simply maps slave IRQ signals to the bits of `irq[31..0]`. Any unassigned bits of `irq[31..0]` are permanently disabled. Figure 2–19 shows an example of the interrupt controller mapping the IRQs on four slave ports to `irq[31..0]` on a master port.

Figure 2–19. IRQ Mapping Using Software Priority

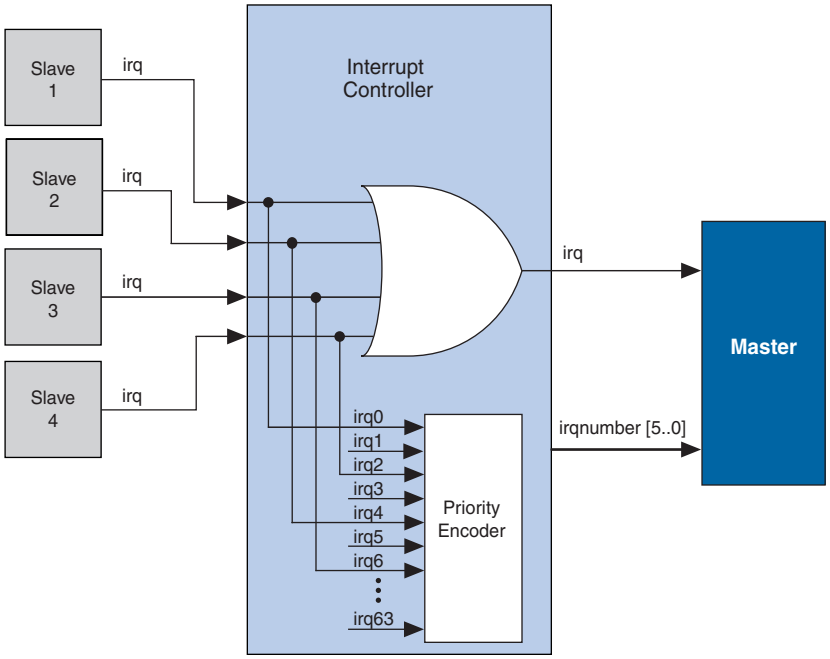


Hardware Priority

In the hardware priority configuration, in the event that multiple slaves assert their IRQs simultaneously, the system interconnect fabric (that is, hardware logic) identifies the IRQ of highest priority and passes only that IRQ number to the master port. An IRQ of lesser priority is undetectable until a master port clears all IRQs of higher priority.

Using hardware priority, the interrupt controller can handle up to 64 slave IRQ signals. The interrupt controller generates a 1-bit `irq` signal to the master port, signifying that one or more slave ports have generated an IRQ. The controller also generates a 6-bit `irqnumber` signal, which outputs the encoded value of the highest pending IRQ. See Figure 2–20.

Figure 2–20. IRQ Mapping Using Hardware Priority

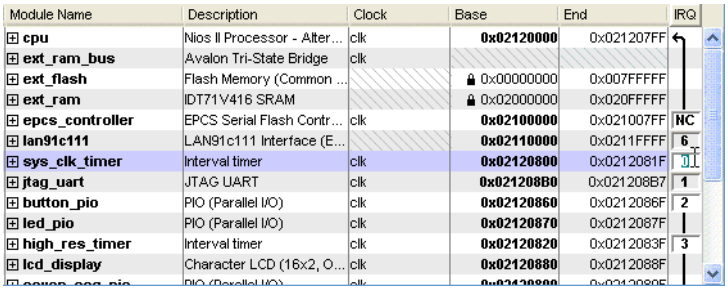


Assigning IRQs in SOPC Builder

You specify IRQ settings on the **System Contents** tab of SOPC Builder. After adding all components to the system, you make IRQ settings for all slave ports that can generate IRQs, with respect to each master port. For

each slave port, you can either specify an IRQ number, or specify not to connect the IRQ. Figure 2–21 shows the IRQ settings for multiple slave IRQs that drive the master component named **cpu**.

Figure 2–21. Assigning IRQs in SOPC Builder



Reset Distribution

The system interconnect fabric generates and distributes a system-wide reset pulse to all logic in the system module. The system interconnect fabric distributes the reset signal conditioned for each clock domain. The duration of the reset signal is at least one clock period.

The system interconnect fabric asserts the system-wide reset in the following conditions:

- The global reset input to the system module is asserted.
- Any slave port asserts its `resetrequest` signal.

All components must enter a well-defined reset state whenever the system interconnect fabric asserts the system-wide reset. The timing of the reset signal is asynchronous to the operation of transfers. Resets are asserted asynchronously and deasserted synchronously to the associated clock.

Referenced Documents

This chapter references the following documents:

- *Avalon Memory-Mapped Interface Specification*
- *System Interconnect Fabric for Streaming Interfaces*
- *Avalon Streaming Interface Specification*
- *Avalon Memory-Mapped Bridges*

Document Revision History

Table 2–4 shows the revision history for this chapter.

Table 2–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Updated to match 7.2 features. Deleted paragraphs discussing “Pipelining for High Performance”, “Endian Conversion”, and added new screenshots. • Moved clock-crossing bridge discussion to this chapter from chapter 10.	—
May 2007, v7.1.0	• Chapter 3 was previously titled Avalon Switch Fabric. • Updated Avalon terminology because of changes to Avalon technologies. Changed old “Avalon switch fabric” term to “system interconnect fabric.” Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface.” • Rearranged content in section “Introduction” on page 2–1 to enhance clarity and to acknowledge the existence of the new Avalon Streaming interface. • In section “Pipelining for High Performance” on page 2–7, noted that automatic pipelining for high performance is a deprecated feature. Added the recommendation to use the Avalon-MM Pipeline Bridge component instead. • Updated Table 2–2 on page 2–9 for improved clarity. • Updated section “Dynamic Bus Sizing” on page 2–9 to reflect new behavior of system interconnect fabric with respect to byte enables during read transfers. For a master-to-slave data-width ratio of *N*:1, the system interconnect fabric might not need to perform *N* slave-side read transfers, depending on how the master port asserts its byte-enable signals. • Added three paragraphs explaining when clock signals are automatically connected to SOPC Builder components. • Added paragraph referencing the higher performance Avalon-MM Clock-Crossing Bridge which can be used instead of the CDC logic for systems requiring higher throughput.	For the 7.1 release, Altera released the Avalon Streaming Interface, which necessitated some re-phrasing of existing Avalon terminology. The newly-released Avalon-MM Pipeline Bridge component provides a more effective means to improve f_{MAX} performance than the traditional pipeline option in SOPC Builder. The behavior of `byteenable` signals in the Avalon Interface Specification was updated, necessitating changes to this document.
March 2007, v7.0.0	No change from previous release.	—
November 2006, v6.1.0	No change from previous release.	—
May 2006, v6.0.0	No change from previous release.	—
October 2005, v5.1.0	No change from previous release.	—
August 2005, v5.0.1	Updated for the Quartus II software version 5.1.	—

Table 2–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
May 2005, v5.0.0	• Added burst transfer management details. • Updated pipeline management details.	—
February 2005, v1.0	Initial release.	—

3. System Interconnect Fabric for Streaming Interfaces

QII54019-7.2.0

Introduction

Avalon® Streaming interconnect fabric connects high-bandwidth, low latency components that use the Avalon Streaming (Avalon-ST) interface. It creates datapaths for unidirectional traffic including multichannel streams, packets, and DSP data. This chapter describes the Avalon-ST interconnect fabric and its use in connecting components with Avalon-ST interfaces. Descriptions of specific adapters and their use in streaming systems can be found in the following sections:

- "Adapters" on page 3–3
- "Multiplexer Examples" on page 3–5

High-Level Description

Avalon-ST interconnect fabric is logic generated by SOPC Builder. Using SOPC Builder, you specify how Avalon-ST source and sink ports connect. SOPC Builder creates a high performance point-to-point interconnect between the two components. The Avalon-ST interconnect is flexible and can be used to implement on-chip interfaces for industry standard telecommunications and data communications cores, such as Ethernet IEEE 802.3 MAC and SPI 4.2. In all cases, bus widths, packets, and error conditions are custom-defined.

Figure 3–1 illustrates the simplest system example that generates an interconnect between the source and sink. This source-sink pair includes only the `data` and `valid` signals.

Figure 3–1. Interconnect for a Simple Avalon Streaming Source-Sink Pair

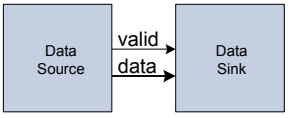
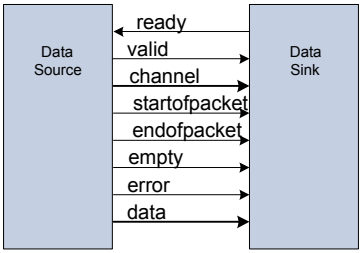


Figure 3–2 illustrates a more extensive interface that includes signals indicating the start and end of packets, channel numbers, error conditions, and back pressure.

Figure 3–2. Avalon Streaming Interface for Packet Data



All data transfers using Avalon-ST interconnect occur synchronously to the rising edge of the associated clock interface. All outputs from the source interface, including the data, channel, and error signals, must be registered on the rising edge of the clock. Registers are not required for inputs at the sink interface. Registering signals at the source provides for high frequency operation while eliminating back-to-back registration with no intervening logic. There is no inherent maximum performance of the interconnect. Throughput for a system depends on the components and how they are connected.

Although you do not have to register signals in the sink-to-source direction, register such signals if more than a trivial amount of logic is needed to generate them. Registering signals at both ends of the source-to-sink connection can increase the f_{MAX} at which the system can run.

For details about the Avalon-ST interface protocol, refer to the *Avalon Streaming Interface Specification* available at **www.altera.com**.

Avalon Streaming and Avalon Memory-Mapped Interfaces

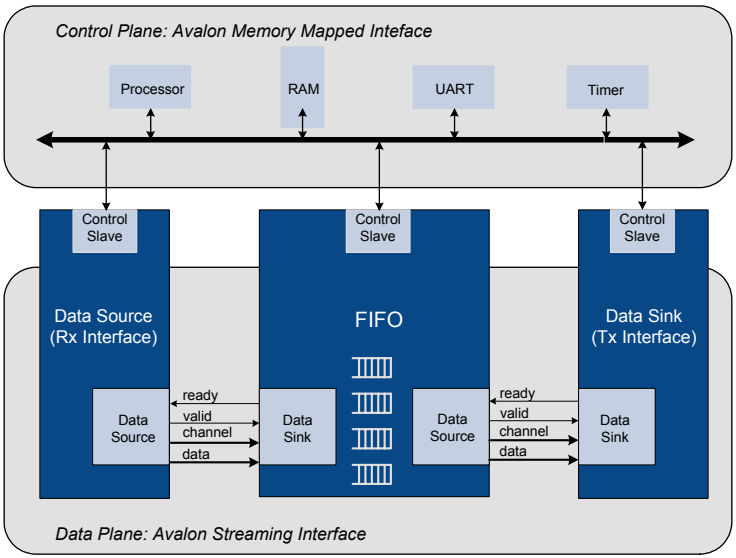
The Avalon-ST and Avalon Memory-Mapped (Avalon-MM) interfaces are complimentary. High bandwidth components with streaming data typically use Avalon-ST interfaces for the high thoughput datapath. These components can also use Avalon Memory-Mapped interfaces to provide an access point for control. In contrast to the Avalon-MM interconnect, which can be used to create a wide variety of topologies, the Avalon-ST interconnect fabric always creates a point-to-point between a single data source and data sink, as Figure 3–3 illustrates. There are two connection pairs in this figure:

- The Data Source in the RX Interface transfers data to the Data Sink in the FIFO.

- The Data Source in the FIFO transfers data to the TX Interface Data Sink.

In Figure 3–3, the Avalon-MM interface allows a processor to access the data source, FIFO, or data sink to provide system control.

Figure 3–3. Use of the Avalon Memory-Mapped and Streaming Interfaces



Adapters

Adapters are configurable SOPC Builder components that are part of streaming interconnect fabric. They are used to connect source and sink interfaces that are not exactly the same without affecting the semantics of the data. SOPC Builder includes the following three adapters:

- Data Format Adapter
- Timing Adapter
- Channel Adapter

The Insert Avalon-ST Adapters command on the System menu allows you to insert an adapter so that you can connect a data source to a data sink of differing byte sizes in the SOPC Builder system.

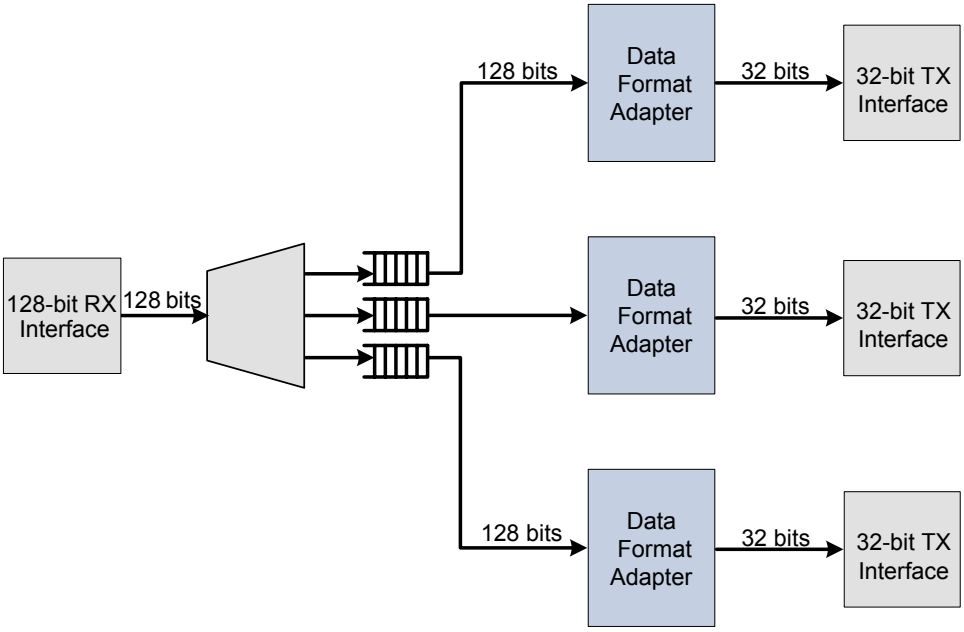
For complete information about these adapters, refer to the *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook*.

The following sections provide an overview of these adapters.

Data Format Adapter

The data format adapter allows you to connect interfaces that have different values for the parameters defining the data signal. One of the most common uses of this adapter is to convert buses of different widths. Figure 3–4 shows an adapter that allows a connection between a 128-bit input bus and three 32-bit output buses.

Figure 3–4. Avalon Streaming Interconnect Fabric with Data Format Adapter



Timing Adapter

The timing adapter allows you to connect component interfaces that require a different number of cycles before driving or receiving data. This adapter inserts a FIFO between the source and sink to buffer data or pipeline stages to delay the backpressure signals. The timing adapter can also be used to connect interfaces that support the `ready` signal and those that do not.

Channel Adapter

The channel adapter provides adaptations between interfaces that have different support for the channel signal or channel-related parameters. For example, if the source channel is narrower than the sink channel, you can use this adapter to connect them. The high-order bits of the sink channel are connected to zero. You can also use this adapter to connect a source with a wider channel to a sink with a narrower channel; however, this usage produces a warning that data may be lost.

Multiplexer Examples

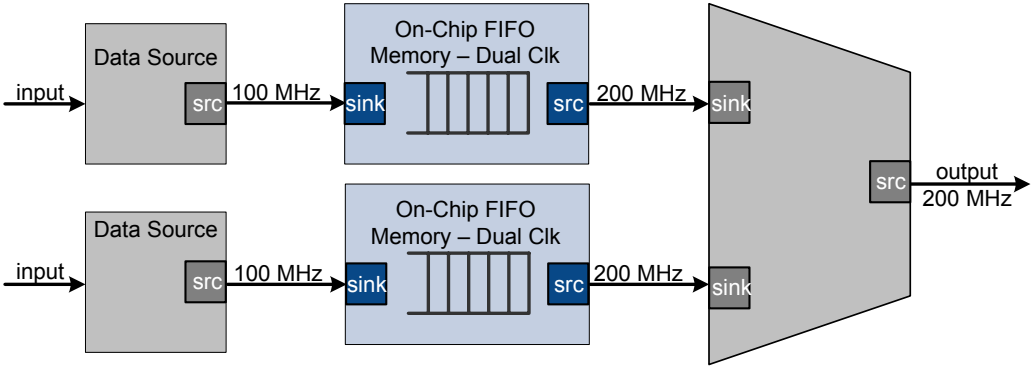
You can combine the three adapters referenced above with streaming components to create datapaths whose input and output streams have different properties. The following sections provide three examples of datapaths constructed using SOPC Builder whose output stream is higher performance than the input stream:

- The first example shows an output with double the throughput of each interface with a corresponding doubling of the clock frequency.
- The second example doubles the data width.
- The third boosts the frequency of a stream by 10% multiplexing input data from 2 sources.

Example to Double Clock Frequency

Figure 3–5 illustrates a datapath that uses the dual clock version of the on-chip FIFO memory and Avalon-ST channel multiplexer to merge the 100 MHz input from two streaming data sources into a single 200 MHz streaming output. As Figure 3–5 illustrates, this example increases throughput by increasing the frequency and combining inputs.

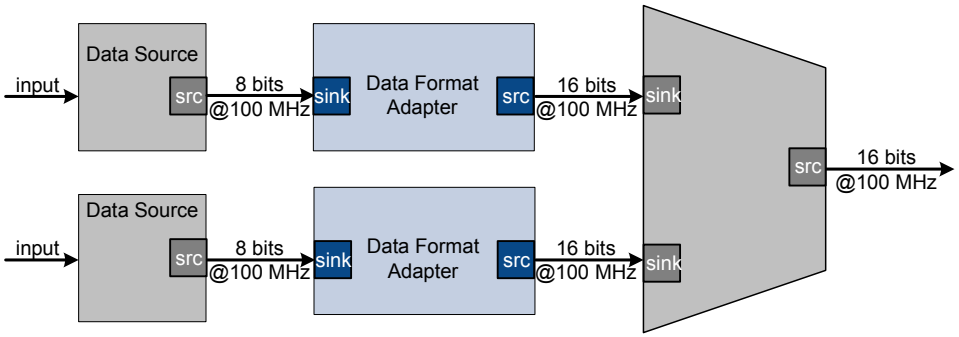
Figure 3–5. Datapath that Doubles the Clock Frequency



Example to Double Data Width and Maintain Frequency

Figure 3–6 illustrates a datapath that uses the data format adapter and Avalon-ST channel multiplexer to convert two, 8-bit inputs running at 100 MHz to a single 16-bit output at 100 MHz.

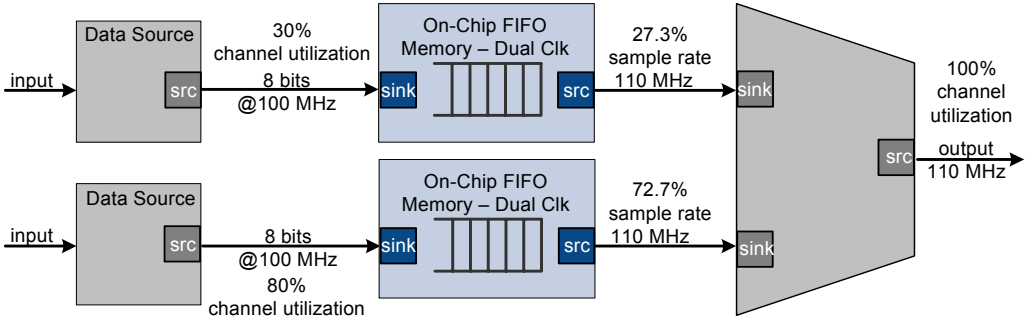
Figure 3–6. Datapath to Double Data Width and Maintain Original Frequency



Example to Boost the Frequency

Figure 3–7 illustrates a datapath that uses the dual clock version of the on-chip FIFO memory to boost the frequency of input data from 100 MHz to 110 MHz by sampling two input streams at differential rates. In this example, the on-chip FIFO memory has an input clock frequency of 100 MHz and an output clock frequency of 110 MHz. The channel multiplexer runs at 110 MHz and samples one input stream 27.3 percent of the time and the second 72.7 percent of the time.

Figure 3–7. Datapath to Boost the Clock Frequency



Referenced Documents

This chapter references the following documents:

- *Avalon Streaming Interface Specification*
- *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook*.

Document Revision History

Table 3–1 shows the revision history for this chapter.

Table 3–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007, v7.2.0	No changes for this release.	—
May 2007, v7.1.0	Initial release.	The Avalon-ST Data Format Adapter, Timing Adapter and Channel Adapter are new components provided in the Quartus II software v7.1 release.

4. SOPC Builder Components

QII54004-7.2.0

Introduction

An SOPC Builder *component* is a hardware design block available within SOPC Builder that can be instantiated in an SOPC Builder system. This chapter defines SOPC Builder components, with emphasis on the structure of custom components.

A component includes the following:

- The HDL description of the component's hardware
- A description of the interface to the component hardware, such as the names and types of I/O signals.
- A description of any parameters that specify the structure of the component logic and component.
- A GUI for configuring an instance of the component in SOPC Builder.
- Scripts and other information SOPC Builder needs to generate the hardware description language (HDL) files for the component and integrate the component instance into the system module.
- Other component-related information, such as reference to software drivers, necessary for development steps downstream of SOPC Builder.

This chapter discusses the design flow for new and legacy custom-defined SOPC Builder components, in the following sections:

- "Component Providers" on page 4–2
- "Component Hardware Structure" on page 4–2
- "List of Available Components in SOPC Builder" on page 4–4
- "Tcl Components" on page 4–5

New Component Structure in v7.1 of the Quartus II Software

Version 7.1 of the Quartus® II software provided a new mechanism for storing and finding component files located on your computer.



If you use components created with a previous version of the Quartus II software, read through this chapter to familiarize yourself with the differences. This document uses the term "legacy components" to refer to components created with a previous version of the Quartus II software.

Legacy components are compatible with newer versions of SOPC Builder, with the following caveats:

- Legacy components that use a **More Options** tab in SOPC Builder, such as complex IP components provided by third-party IP developers, cannot be instantiated or used in version 7.1 and beyond. If your component has a "bind" program, you cannot use the component.
- To edit a legacy component using the component editor in version 7.1 and beyond, you must first upgrade the component to the new component editor flow. The process is automatic. However, the result is not backward compatible with previous versions.

Component Providers

SOPC Builder components can be installed on your computer by several possible providers, including the following:

- The Quartus II software, which includes SOPC Builder, can install components as part of the fundamental functionality of the software.
- The Altera® MegaCore IP Library provides several intellectual property (IP) design blocks that are SOPC Builder ready.
- Third-party IP developers can provide IP blocks as SOPC Builder ready components, including software drivers and documentation.
- Altera development kits, such as the Nios® II Development Kit, can provide SOPC Builder components as features.
- The SOPC Builder component editor can turn your own HDL files into custom components.

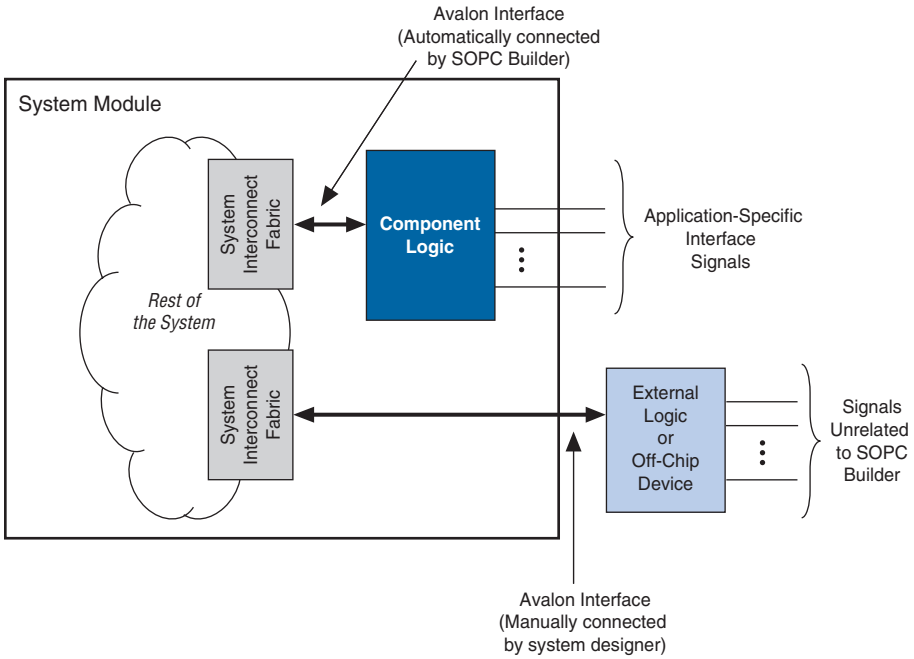
Component Hardware Structure

There are two types of components, based on where the associated component logic resides:

- Components that include their associated logic inside the system module
- Components that interface to logic outside the system module

Figure 4–1 shows an example of both types of components.

Figure 4–1. Component Logic Inside and Outside the System Module



Components That Include Logic Inside the System Module

For components that include logic inside the system module, the component provides a full description of its hardware by specifying an HDL file. During system generation, SOPC Builder instantiates the component in the system and connects it to the rest of the system. The component can include export signals, which become ports on the system itself, so that you can manually connect them to logic outside the system module.

In general, components connect to the system interconnect fabric using either the Avalon® Memory-Mapped (Avalon-MM) interface or the Avalon Streaming (Avalon-ST) interface. A single component can provide more than one Avalon port. For example, a component might provide an Avalon-ST source port for high-throughput data, in addition to an Avalon-MM slave port for control.

Components That Interface to Logic Outside the System Module

For components that interface to logic outside the system module, the component files describe only the interface to the external logic. During system generation, SOPC Builder only exports an interface for the component to the top-level system module. You must manually connect the interface to the component outside the system.

List of Available Components in SOPC Builder

Each time SOPC Builder starts, it searches for component files. The components that SOPC Builder finds are displayed in the list of available components on the SOPC Builder **System Contents** tab. There are several mechanisms that SOPC Builder uses to populate the list of available components:

- SOPC Builder automatically searches the **/ip** subdirectory of your Quartus II project directory. Adding a component to a project is as easy as copying it to a subdirectory here. This mechanism is recommended for all project-specific components.
- SOPC Builder searches all of the paths entered in **SOPC Builder/Tools/Options/IP Search Path** to support a global library of components. This mechanism is recommended for all global components.
- Quartus II project directory and user library paths—SOPC Builder identifies component files stored in the current Quartus II project directory and user library paths.
- Legacy component search paths—SOPC Builder searches the paths where previous versions of SOPC Builder expected to find component files.

The rest of this section focuses on Tcl components.

Tcl Components

Tcl components are components where interaction with SOPC Builder is defined with a simple text file written in the Tcl scripting language. This section describes the structure of Tcl components and how they are stored.



For details on the SOPC Builder component editor, refer to the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*.

Component Description File (_hw.tcl)

At a minimum, a Tcl component consists of the following files:

- A Verilog, HDL, or VHDL file that defines the top-level module of the custom component (optional).
- A component description file, which is a Tcl file with file name of the form **<*entity name*>_hw.tcl.**

The **_hw.tcl** file defines everything that SOPC Builder requires about the name and location of component design files.

The SOPC component editor can generate components without Verilog HDL or VHDL files.

Component File Organization

A typical component uses the following directory structure. The names of the directories are not significant.

- **component_library/**
  - **hdl/**— a directory that contains the component HDL design files and the **_hw.tcl** file
    - **<*component name*>_hw.tcl**—the component description file
    - **<*component name*>.v or .vhd**—the HDL file that contains the top-level module
- There is no expectation of an HDL folder, even for components that are created with the component editor. If you want to bundle your component in a directory, the basic structure is as follows:
    - ***component_dir/***
    - **<*name*>_hw.tcl**
    - **<*name*>.v or .vhd**
    - **<*name*>_sw.tcl**
- **software/**—a directory that contains software drivers or libraries related to the component, if any

The component directory will often include a **_sw.tcl** file and the software definitions and drivers it refers to. Refer to the component software specification for further details.

Referenced Document

This chapter references Chapter 5, Component Editor.

Document Revision History

Table 4–1 shows the revision history for this chapter.

Table 4–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007, v7.2.0	• Description added of Tcl components and removal of custom-defined components. • Added warning that SOPC Builder does not support parameter values > 31 bits	—
May 2007, v7.1.0	• Described the new structure of components which is new in 7.1. • Added and updated the sources of components list. • Reorganized content of the chapter. • Updated Avalon terminology because of changes to Avalon technologies. Changed old "Avalon switch fabric" term to "system interconnect fabric." Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface." • Removed description of SOPC Builder MegaWizard® Plug-In Manager component discovery mechanism that was inaccurate.	Version 7.1 of the Quartus II software provides a new mechanism for storing and finding SOPC Builder component files located on your computer, which necessitates significant changes to this chapter.
March 2007, v.7.0.0	No change from previous release.	—
November 2006, v.6.1.0	No change from previous release.	—
May 2006, v6.0.0	No change from previous release.	—
October 2005, v5.1.0	No change from previous release.	—
August 2005, v5.0.1	Corrected reference to figure.	—
May 2005, v5.0.0	No change from previous release.	—
February 2005, v1.0	Initial release.	—

5. Component Editor

QII54005-7.2.0

Introduction

This chapter describes the SOPC Builder component editor. The component editor provides a GUI to support the creation and editing of the **_hw.tcl** file that describes a component to SOPC Builder. You use the component editor to do the following:

- Specify a hardware description language (HDL) file that describes the modules that compose your component hardware.
- Define the interfaces on the component and provide information about how the interface functions.
- Specify the hardware interface or interfaces to the component, and define the behavior of each interface signal. Assign module signals to interfaces and determine signal roles.
- Specify relationships between interfaces, such as determining which clock interface is used by a slave interface.
- Declare any parameters that alter the component structure or functionality, and define a user interface to let users parameterize instances of the component.

For information on the use of the component editor, see the following sections:

- To start the component editor, refer to "Starting the Component Editor" on page 5–2.
- For information about specifying HDL files that describe a component, refer to "HDL Files Tab" on page 5–2.
- For information about specifying interface signals, refer to "Signals Tab" on page 5–3.
- For information about specifying the Avalon-MM type of interface signals, refer to "Interfaces Tab" on page 5–6.
- For information about specifying parameters, refer to "Component Wizard Tab" on page 5–6.
- To save a component, refer to "Saving a Component" on page 5–7.
- For information about changing a component after it has been saved, refer to "Editing a Component" on page 5–8.



For more information about components, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*. For more information about the Avalon-MM interface, refer to the *Avalon Memory-Mapped Interface Specification*.

Component Hardware Structure

The component editor creates components with the following hardware characteristics:

- A component has one or more interfaces. Typically, an *interface* means an Avalon-MM master port or slave port. The component editor lets you build a component with any combination of Avalon-MM master or slave ports. You can also specify component signals that must appear at the top-level of the SOPC Builder system module, which you can manually connect to the logic outside the system module. Interfaces include:
  - Avalon-MM master/slave
  - Avalon Streaming source/sink
  - Interrupt sender/receiver
  - Clock input and output
  - Nios II Custom Instruction Conduit (for export only)
- Each interface is comprised of one or more signals.
- The component can represent a component that is instantiated inside the SOPC Builder system, and can represent a component outside the system with an interface to it on the generated system.

Starting the Component Editor

To start the component editor in SOPC Builder, on the File menu, click **New Component**. When the component editor starts, the **Introduction** tab displays, which describes how to use the component editor.

The component editor presents several tabs that group related settings. A message window at the bottom of the component editor displays warning and error messages.



Each tab in the component editor provides on-screen information that describes how to use the tab. Click the triangle labeled **About** at the top-left of each tab to view these instructions. You can also refer to Quartus® II Online Help for additional information about the component editor.

You navigate through the tabs from left to right as you progress through the component creation process.

HDL Files Tab

The first row of the table on the **HDL Files** tab must include the file with the top-level module and must specify all the HDL files.You use the **HDL Files** tab to specify an existing Verilog HDL, or VHDL file that describes the interface to the component hardware. If your component is an interface to external logic, then do not specify an HDL file.

You can also use the component editor to define logic interfaces to external logic. In this case, you do not provide HDL files, and instead you use the component editor to manually define the hardware interface.

After you specify an HDL file, the component editor immediately analyzes the file by invoking the Quartus II Analysis and Elaboration module. The component editor analyzes signals and parameters declared for all modules in the specified files. If the file is successfully analyzed, the component editor's **Signals** tab lists all design modules in the **Top Level Module** list. If your HDL contains more than one module, you must select the appropriate top-level module from the **Top Level Module** list.

If your design requires extra simulation files, you can specify them in the **Simulation Files** table. All files used in the simulation must be specified, even those already included for synthesis. SOPC Builder includes these files in the system test bench so they can provide special functionality during simulation. The simulation files do not affect the generated system hardware.



When the top-level module is changed, the component editor performs best-effort signal matching against the existing port definitions. If a port is absent from the module, it is removed from the port list.

Signals Tab

You use the **Signals** tab to specify the purpose of each signal on the top-level component module. If you specified a file on the **HDL Files** tab, the signals on the top-level module appear on the **Signals** tab.

If the component is an interface to external logic, you must manually add the signals that comprise the interface to the external logic. The **Interface** list also allows creation of a new interface.

Each signal must belong to an interface and be assigned a signal type. The signal type for new signals that have not been assigned a signal type is `Export`, which means that SOPC Builder does not connect the signal internally to the system module, and instead exposes the signal on the top-level system module.

You assign each signal to an interface using the **Interface** list. In addition to Avalon Memory-Mapped and Streaming interfaces, components typically have a conduit interface for exported signals.

Naming Signals for Automatic Type and Interface Recognition

The component editor recognizes signal types and interfaces based on the names of signals in the source HDL file, if they follow naming conventions. Table 5–1 lists the signal naming conventions.

Table 5–1. Conventions of Automatically Recognized Signal Names

Type of Signal	Name Convention
Signal associated with a specific interface	*<interface type>*_*<interface name>*_*<signal type>*[_n]

For any value of Interface Name the component editor automatically creates an interface by that name, if necessary, and assigns the signal to it. The *Signal Type* must match one of the valid signal types for the type of interface. You can append `_n` to indicate an active-low signal. Table 5–2 lists the valid values for Interface Type.

Table 5–2. Valid Values for <Interface Type>

Value	Meaning
avs	Avalon-MM slave
avm	Avalon-MM master
ats	Avalon-MM tristate slave
atm	Avalon-MM Tristate Master
aso	Avalon-ST Source
asi	Avalon-ST Sink
cso	Clock Output
csi	Clock Input
inr	Interrupt Receiver
ins	Interrupt Sender
cos	Conduit Start
coe	Conduit End
ncm	Nios II Custom Instruction Master
ncs	Nios II Custom Instruction Slave
csi_clockreset_clk	Clock Reset
csi_clockreset_reset_n	Clock Reset N

Example 5–1 shows a Verilog HDL module declaration with signal names that infer two Avalon-MM slave ports.

Example 5–1. Verilog Module With Automatically Recognized Signal Names

```
module my_multiport_component (
       // Signals for Avalon-MM slave port "s1"
       avs_s1_clk,
       avs_s1_reset_n,
       avs_s1_address,
       avs_s1_read,
       avs_s1_write,
       avs_s1_writedata,
       avs_s1_readdata,
       avs_s1_export_dac_output,

       // Signals for Avalon-MM slave port "s2"
       avs_s2_address,
       avs_s2_read,
       avs_s2_readdata,
       avs_s2_export_dac_output,

       // Clock/Reset Interface csi_clockreset_clk
   );
```

Templates for Interfaces to External Logic

If you are creating an interface to external logic, you can use the Templates menu in the component editor to add a set of signals, such as the following:

- Avalon-MM Slave
- Avalon-MM Slave with Interrupt
- Avalon-MM Master
- Avalon-MM Master with Interrupt
- Avalon-ST Source
- Avalon-ST Sink

After adding a template, you can add or delete signals to customize the interface to meet your needs.

Interfaces Tab

The **Interfaces** tab allows you to configure the interfaces on your component, and specify a name for each interface. The interface name identifies the interface, and also appears in the SOPC Builder connection panel. The interface name is also used to uniquely identify any signals that are exposed on the top-level system module.

The **Interfaces** tab also allows you to configure the type and properties of each interface. For example, an Avalon-MM slave interface has timing parameters which you must set appropriately.

If you convert an older Avalon-MM slave to the new model, you may require three interfaces: a clock input, the Avalon slave, and an interrupt sender. A parameter in the interrupt sender must be set to reference the Avalon slave.

Component Wizard Tab

The **Component Wizard** tab provides options that affect the presentation of your new component.

Identifying Information

You can specify information that identifies the component as follows:

- **Folder**—Specifies the location of the component, determined by the location of the top-level HDL file.
- **Component Display Name**—Specifies the internal name of the component. The internal name is used when saving a system containing an instance of this component, and is the name use for the component type when you create a system using a script..
- **Component Version**—Specifies which version of the component you are using.
- **Component Group**—Specifies which group in SOPC Builder displays your component in the list of available components. If you enter a previously unused group name, SOPC Builder creates a new group by that name.
- **Description**—Allows you to describe the component (optional).
- **Created By**—Allows you to specify the author of the component (optional).
- **Icon**—Allows you to associate the component with a file path relative to the component. The icon can be a **.jpg**, **.gif**, or **.png** file (optional).
- **Parameters**—Allows you to specify the parameters for creating the component. See further description below.

The component editor assigns the class name to be the same name as the top-level HDL module. The class name is the name SOPC Builder uses to identify the component.

Parameters

The **Parameters** table allows you to specify the user-configurable parameters for the component.

If the top-level module of the component HDL declares any parameters (*parameters* for Verilog, HDL, or *generics* for VHDL), those parameters appear in the **Parameters** table. These parameters are presented to you when you create or edit an instance of your component. Using the **Parameters** table, you can specify whether or not each parameter is user-editable.

The following rules apply to HDL parameters exposed via the component GUI:

- Editable parameters cannot contain computed expressions.
- If a parameter *N* defines the width of a signal, the signal width must be of the form *N*-1..0.
- When a VHDL component is used in a Verilog HDL system module, or vice versa, numeric parameters must be 32-bit decimal integers. Passing other numeric parameter types might fail.

Click **Preview the Wizard** at any time to see how the component GUI will appear.

Saving a Component

You can save the component by clicking **Finish** on any of the tabs, or by clicking **Save** on the File menu. Based on the settings you specify in the component editor, the component editor creates a component description file with the file name ***<name of top-level module>*_hw.tcl**. The component editor saves the file in the same directory as the HDL file that describes the component's hardware interface. If you did not specify an HDL file, you can save the component description file to any location you choose.

You can relocate component files later. For example, you could move component files into a subdirectory and store it in a central network location so that other users can instantiate the component in their systems.

Editing a Component

After you save a component and exit the component editor, you can edit it in SOPC Builder. To edit a component, right-click it in the list of available components on the **System Contents** tab and click **Edit Component**.



You cannot edit components that were created outside of the component editor, such as Altera®-provided components.

If you edit the HDL for a component and change the interface to the top-level module, you need to edit the component with the component editor to reflect the changes you made to the HDL.

Referenced Documents

This chapter references the following documents:

- *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*
- *Avalon Memory-Mapped Interface Specification*
- *Building a Component Interface with TCL Scripting Commands* chapter in volume 4 of the *Quartus II Handbook*
- *Nios II Software Developer's Handbook*

Document Revision History

Table 5–3 shows the revision history for this chapter.

Table 5–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007, v7.2.0	• Updated several paragraphs describing the latest GUI.	—
May 2007, v7.1.0	• Updated all sections to reflect significant functional differences in version 7.1. • Added section "Changes to Component Editor in Version 7.1" on page 5–2. • Updated section "Component Editor Output" and "Re-editing Components" to accommodate new component structure with 7.1 release. • Updated Avalon terminology because of changes to Avalon technologies. Changed old "Avalon switch fabric" term to "system interconnect fabric." Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface." • Removed screen shots that simply reflect what user sees when using the tool without illustrating a particular point. • Added Referenced Documents section which links to all referenced documents. • Added statement that all simulation files, not just top-level file, must be added using the HDL files tab.	The file structure of SOPC Builder components changed significantly in this release, which required substantial functional change to the component editor. This document changed significantly to reflect the functional changes. Updated to improve readability.
March 2007, v7.0.0	No change from previous release.	—
November 2006, v6.1.0	No change from previous release.	—
May 2006, v6.0.0	No change from previous release.	—
December 2005, v5.1.1	• Added section "Naming Signals for Automatic Type and Interface Recognition" on page 5–4. • Added section "Templates for Interfaces to External Logic" on page 5–6. • Clarified operation of the Save command. • Updated all screenshots.	—
October 2005, v5.1.0	No change from previous release.	—
May 2005, v5.0.0	Initial release.	—

6. Building a Component Interface with Tcl Scripting Commands

QII54022-7.2.0

This chapter describes the Tcl scripting commands that you can use to define custom components for use in an SOPC Builder system. You can also use the scripting interface to declare and set parameter values for your components.

The Tcl scripting commands provide a programmatic interface that you might prefer to the graphical user interface (GUI) of the component editor. If you need to make global updates to multiple components, Tcl scripts allow you to make the changes without accessing each component through the GUI.

You can use the Tcl scripting commands or the component editor to create a component description file with the file name ***<name of top-level module>*_hw.tcl**. This file is stored in the same directory as the HDL file that provides the top-level description of the component. You can edit this file using the text editor of you choice.

You can download sample ***_hw.tcl** files from the Altera website by clicking the Design Example hyperlink located under this chapter, *Building a Component Interface with Tcl Scripting Commands*.

The remainder of this chapter describes the commands and properties you can use to describe components, component interfaces and parameters. These include:

- "Organization of a Component Tcl File" on page 6–2
- "Set and Add Commands" on page 6–3
- "Module Properties" on page 6–4
- "Clock Interface" on page 6–4
- "Avalon-MM Master Interface" on page 6–5
- "Avalon-MM Slave Interface" on page 6–5
- "Avalon-MM Tristate Interface" on page 6–7
- "Nios II Custom Instruction Interface" on page 6–8
- "Interrupt Interface" on page 6–9
- "Conduit Interface" on page 6–10

Organization of a Component Tcl File

The following steps describe how to organize a component Tcl file.

1. Start the component definition with the `set_source` command, followed by the `set_module` command. The name of the module must match the component's top-level Verilog or VHDL entity name.

Example 6–1. Example of Set Module Command

```
set_module "my_module"
```

2. Define the module properties, which are pieces of static information about a module. The following example illustrates some of the `set` command and `module` properties. See Table 6–5.

Example 6–2. The Set Command and Module Properties

```
set_source_file "./my_component.v"
set_module_description "My Component"
set_module_property version "1.0"
set_module_property group "My Components"
set_module_property simulationFiles [ list "./my_component.v" ]
```

3. Define the module parameters, which are settings that the user of the component makes when parameterizing it. The following example illustrates how to define module parameters.

Example 6–3. Example of Parameters

```
# Module parameters
add_parameter "DWIDTH" "integer" "32" ""
add_parameter "AWIDTH" "integer" "32" ""
```

4. Add interfaces. For each interface, first add the interface, then set its properties and define its ports. Refer to the Avalon-MM specification for port types. The following example defines an Avalon-MM slave interface using only the required properties.

Example 6–4. Avalon-MM Slave Interface

```
# Interface my_slave
# all interfaces must specify an associated clock interface
add_interface "my_slave" "avalon" "slave" "my_clock_interface"

set_interface_property "my_slave" "timingUnits" "cycles"
```

```
set_interface_property "my_slave" "writeWaitTime" "0"
set_interface_property "my_slave" "readLatency" "0"
set_interface_property "my_slave" "holdTime" "0"
set_interface_property "my_slave" "readWaitTime" "0"
set_interface_property "my_slave" "setupTime" "0"

# Ports in interface my_slave
add_port_to_interface "my_slave" "my_slave_write" "write"
add_port_to_interface "my_slave" "my_slave_writedata" "writedata"
add_port_to_interface "my_slave" "my_slave_waitrequest" "waitrequest"
```

Set and Add Commands

The `set` and `add` commands establish basic information about a component.

Table 6–1. Set and Add Commands

Command	Arguments
`set_module`	*<name of the module> (1)*
`set_source_file`	*<path to HDL file> (2)*
`set_module_description`	*<description of the module>*
`set_module_property`	*<name of property> <value of property>*
`add_interface`	*<name of interface> <type of interface> <direction> <associated clock>(3)*
`set_interface_property`	*<name of interface> <name of property> <value of property>*
`add_port_to_interface`	*<name of interface> <port name> <type of port>*
`set_port_direction_and_width`	*<name of port> <direction> <width>*

Notes to Table 6–1:

(1) Declares a new module. Must match the top-level Verilog HDL module or VHDL entity.

(2) If the component is not based on HDL, `set_source_file` should be used with an empty string, such as "`set_source_file`".

(3) This command is only required when a source file is not set. If a source file is set, the Quartus II software analyzes the file and determines the port widths and directions.

Module Properties

The module properties are the arguments to the `set_module_property` command. Table 6–2 lists the module properties.

Table 6–2. Module Properties

Name	Legal Values	Description
`version`	dotted integers	A version string, for example: 1.2.3
`group`	string	A string that represents the category under which the component should be listed.
`simulationFiles`	list of strings	The name of HDL files for use in simulation. This parameter is required even if the same file is used for synthesis and simulation. All files required for simulation must be specified, not just the top-level file.
`synthesisFiles`	list of strings	The name of HDL files for use in synthesis.
`author`	string	Name of the component author.
`iconPath`	string	Path to an image file, which contains an icon to show in the default editor. When referring to local files, they are relative to the Tcl File (**.tcl**).
`datasheetURL`	string	URL pointing to the component datasheet. Can be local or on a network. When referring to local files, they are relative to the TCL file.

Clock Interface

There are no special properties for clock interfaces. A clock interface should not specify an associated clock interface. Clock interface directions are "source" and "input". The following example defines a clock interface.

Example 6–5. Clock Interface

```
# Clock Interface <my_clk_interface>
add_interface "my_clk_interface" "clock" "input"
set_interface_property "clock" "externallyDriven" "false"
set_interface_property "clock" "clockRateKnown" "false"
set_interface_property "clock" "clockRate" "0"
# Ports in interface clock
add_port_to_interface "clock" "clk" "clk"
add_port_to_interface "clock" "reset_n" "reset_n"
```

Avalon-MM Master Interface

Table 6–3 describes the properties that characterize an Avalon-MM master interface. The direction of an Avalon-MM master interface is "master".

Table 6–3. Avalon-MM Master Interface Properties

Name	Default Value	Legal Values	Description
doStreamReads	false	(true,false)	Specifies whether the master supports Avalon flow control read accesses. (This propertry is optional).
doStreamWrites	false	(true,false)	Specifies whether the master supports Avalon flow control write accesses. (This property is optional).—
burstOnBurstBoundaries Only	false	(true,false)	If true, bursts are aligned on burst size. (This property is optional.)

Avalon-MM Slave Interface

Table 6–4 describes the properties that characterize an Avalon-MM slave interface. The direction of an Avalon-MM slave interface is "slave".

Table 6–4. Avalon-MM Slave Interface Properties (Part 1 of 2)

Name	Default Value	Legal Values	Description
readLatency	0	[0 - 63]	Read latency for fixed-latent slaves.
timingUnits	cycles	(cycles, nanoseconds)	Specifies the unit for writeWaitTime, readWaitTime.
writeWaitTime	0	[1000 - 0]	Specifies additional time in units of timeUnits for write to be asserted.
holdTime	0	-	Specifies time in timeUnits between deassertion of read/write and deassertion of chipselect, address and data.
readWaitTime	1	[1000 - 0]	Specifies additional time in units of timeUnits for read to be asserted.
setUpTime	0	[1000 - 0]	Specifies time in timeUnits between assertion of chipselect, address and data and assertion of read/write.
maximumPendingReadTransactions	0	position	The maximum number of pending read accesses which can be queued up by the slave.
burstOnBurstBoundaries Only	false	(true,false)	If true, bursts are aligned on burst size.

Table 6–4. Avalon-MM Slave Interface Properties (Part 2 of 2)

Name	Default Value	Legal Values	Description
isNonVolatileStorage	false	(true,false)	For software environment purposes. Indicates if the memory is a non-volatile storage device.
printableDevice	false	(true,false)	For software environment purposes. Indicates if the memory is a non-volatile storage device.
isMemoryDevice	false	(true,false)	For software environment purposes. States that the slave is a reasonable target for code and data.

Avalon-ST Source Interface

Table 6–5 lists the properties that characterize an Avalon-ST source interface. Refer to the Avalon-ST specification for port types. The direction of an Avalon-ST source interface is "source".

Table 6–5. Avalon-ST Source Interface Properties

Name	Default Value	Legal Values	Description
symbolsPerBeat	1	[1-512]	The number of symbols that are transferred on every valid cycle.
dataBitsPerSymbol	8	[1-512]	Defines the number of bits per symbol. Most interfaces are byte-oriented so that a symbol is 8 bits.
readyLatency	0	[8-0]	Defines the relationship between assertion/deassertion of the ready signal, and cycles which are considered to be ready for data transfer, separately for each interface.
maxChannel	0	[low-high]	The maximum number of channels that a data interface can support.

Avalon-ST Sink Interface

Table 6–6 lists the properties that characterize an Avalon-ST sink interface. Refer to the Avalon-ST specification for port types. The direction of an Avalon-ST sink interface is "sink".

Table 6–6. Avalon-ST Sink Interface Properties

Name	Default Value	Legal Values	Description
symbolsPerBeat	1	[512-1]	The number of symbols that are transferred on every valid cycle.
dataBitsPerSymbol	8	[512-1]	Defines the number of bits per symbol. Most interfaces are byte-oriented so that a symbol is 8 bits.
readyLatency	0	[8-0]	Defines the relationship between assertion/deassertion of the ready signal, and cycles which are considered to be ready for data transfer, separately for each interface.
maxChannel	0	[255-0]	The maximum number of channels that a data interface can support.

Avalon-MM Tristate Interface

Table 6–7 lists the properties that characterize an Avalon-MM tristate interface. The Avalon-MM tristate interface properties include all the properties that define the Avalon-MM slave interface, plus two additional properties: activeCSThroughReadLatency and maximumPendingReadTransactions.



Note that maximumPendingReadTransactions is not tristate specific. This property can also be assigned to an Avalon State.

The direction of an Avalon-MM tristate interface is "slave".

Table 6–7. Avalon-MM Tristate Interface Properties (Part 1 of 2)

Name	Default Value	Legal Values	Description
readLatency	0	num_cycles	Read latency for fixed-latency slaves.
writeLatency	0	num_cycles	Delay in cycles between acceptance of a write access and acceptance of valid writedata.
timingUnits	cycles	(cycles, nanoseconds)	Specifies the unit for writeWaitTime and readWaitTime.
writeWaitTime	0	[1000-0]	Specifies additional time in units of timeUnits for write to be asserted.

Table 6–7. Avalon-MM Tristate Interface Properties (Part 2 of 2)

Name	Default Value	Legal Values	Description
holdTime	0	—	Specifies time in timeUnits between deassertion of read/write and deassertion of chipselect, address and data.
readWaitTime	1	[1000-0]	Specifies additional time in units of timeUnits for read to be asserted.
setupTime	0	—	Specifies time in timeUnits between assertion of chipselect, address, and data and assertion of read/write.
activeCSThroughReadLatency	false	(true,false)	If true, assert chipselect while readdata is pending.
maximumPendingReadTransactions	false	—	States the maximum number of pending read transactions.
minimumUninterruptedRunLength	1	an integer	Specifies a minimum arbitration share value.
isNonVolatileStorage	false	(true,false)	For software environment purposes. True for flash memories.
printableDevice	false	(true,false)	For software environment purposes. States that the slave is a reasonable sink for printf() data.
isMemoryDevice	false	(true,false)	For software environment purposes. States that the slave is a reasonable target for code and data.

Nios II Custom Instruction Interface

Table 6–8 lists all the properties that characterize Nios II custom instructions.

Table 6–8. Nios II Custom Instruction Interface

Name	Default Value	Legal Values	Description
operands	0	[2-0]	Number of operands used by the custom instruction module.
clockCycle	0	—	Number of clock cycles the custom instruction requires before a valid result is returned—used by multicycle custom instructions.

The following example illustrates all the properties for a custom instruction.

Example 6–6. Custom Instruction Example

```
set_source_file "custominstruction.v"
set_module "custominstruction"
set_module_description "A custom instruction"
set_module_property version "1.0"
set_module_property group "User Logic"

# Module parameters
# Interface nios_custom_instruction_slave_0
add_interface "nios_custom_instruction_slave_0" "nios_custom_instruction" "slave"
"asynchronous"
set_interface_property "nios_custom_instruction_slave_0" "operands" "2"
set_interface_property "nios_custom_instruction_slave_0" "clockCycle" "2"

# Ports in interface nios_custom_instruction_slave_0
add_port_to_interface "nios_custom_instruction_slave_0" "clk" "clk"
add_port_to_interface "nios_custom_instruction_slave_0" "reset" "reset"
add_port_to_interface "nios_custom_instruction_slave_0" "clk_en" "clk_en"
add_port_to_interface "nios_custom_instruction_slave_0" "start" "start"
add_port_to_interface "nios_custom_instruction_slave_0" "n" "n"
add_port_to_interface "nios_custom_instruction_slave_0" "dataa" "dataa"
add_port_to_interface "nios_custom_instruction_slave_0" "datab" "datab"
add_port_to_interface "nios_custom_instruction_slave_0" "a" "a"
add_port_to_interface "nios_custom_instruction_slave_0" "b" "b"
add_port_to_interface "nios_custom_instruction_slave_0" "c" "c"
add_port_to_interface "nios_custom_instruction_slave_0" "readra" "readra"
add_port_to_interface "nios_custom_instruction_slave_0" "readrb" "readrb"
add_port_to_interface "nios_custom_instruction_slave_0" "writerc" "writerc"
add_port_to_interface "nios_custom_instruction_slave_0" "result" "result"
add_port_to_interface "nios_custom_instruction_slave_0" "done" "done"
```

Interrupt Interface

Slave components in an SOPC Builder system typically generate interrupts. A processor typically clears the interrupt bits in the slave's `control` and `status` registers after servicing the interrupt. Table 6–9 lists the properties that characterize interrupts. The direction of an interupt interface is "sender" and "receiver".

Table 6–9. Interrupt Interface Properties

Name	Default Value	Legal Values	Description
`associatedAddressablePoint`	—	an interface name	This parameter takes the name of the component interface that provides access to the registers that should be cleared after the interrupt is serviced.

The following example defines an interrupt interface.

Example 6–7. Interrupt Interface

```
# IRQ Interface my_slave_irq
# legal values for the third parameter <direction> are sender and receiver
add_interface my_slave_irq "interrupt" "sender" "global_signals_clock"

set_interface_property "my_slave_irq" "associatedAddressablePoint" "my_slave"

# Ports in interface my_slave_irq
# Generally there is only one signal of type interrupt
add_port_to_interface "my_slave_irq" "my_irq" "irq"
```

Conduit Interface

A conduit interface is used to export arbitrary input and output signals outside of an SOPC Builder system. There are no special properties associated with conduit interfaces.

The following example illustrates the conduit interface.

Example 6–8. Conduit nterface

```
# Wire Interface global_signals_export
add_interface "global_signals_export" "conduit" "output" "my_clk_interface"

# Ports in interface global_signals_export
add_port_to_interface "global_signals_export" "prbs_test_error" "export"
add_port_to_interface "global_signals_export" "prbs_test_done" "export"
```

Document Revision History

Table 6–10 shows the revision history for this chapter.

Table 6–10. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007, v7.2.0	Major reorganization of chapter to better reflect work flow when using tcl scripting. Includes new commands, properties, and parameters.	—
May 2007, v7.1.0	Initial release.	—

7. Archiving SOPC Builder Projects

QII54017-7.2.0

Introduction

This chapter helps you identify the files you must include when archiving an SOPC Builder project. With this information, you can archive:

- The SOPC Builder system module
- The associated Nios® II software project, if any
- The associated Nios II system library project, if any

You may want to archive your SOPC Builder system for one of the following reasons:

- To place an SOPC Builder design under source control
- To create a backup
- To bundle a design for transfer to another location

To use this information, you must decide what source control or archiving tool to use, and you must know how to use it. This chapter does not provide step-by-step instructions. It does cover the following information:

- How to find and identify the files that you must include in an archived SOPC Builder design, refer to "Required Files" on page 7–2.
- Which files must have write permission to allow the design to be generated and the software projects compiled, refer to "File Write Permissions" on page 7–4.

Scope

This chapter provides information about archiving SOPC Builder system modules, including their Nios II software applications, if any. If your SOPC Builder system does not contain a Nios II processor, you can disregard information about Nios II software applications.

This chapter does not cover archiving SOPC Builder components, for two reasons:

- SOPC Builder components can be recovered, if necessary, from the original Quartus® II and Nios II installations.
- If your SOPC Builder system was developed with an earlier version of the Quartus II software and Nios II Embedded Design Suite (EDS), when you restore it for use with the current version, you normally use the current, installed components.

If your SOPC Builder system was developed with an earlier version of the Quartus II and Nios II development software and you restore it for use with the current version, the regenerated system is functionally identical to the original system. However, there might be differences in details such as Quartus II timing, component implementation, or HAL implementation.



For details of version changes, refer to the release notes for the Quartus II software and the Nios II EDS.

To ensure that you can regenerate your exact original design, maintain a record of the tool and IP version(s) originally used to develop the design. Retain the original installation files or media in a safe place.

The archival process addressed by this chapter is different than Quartus II project archiving. A Quartus II project archive contains the complete Quartus II project, including the SOPC Builder module, but not including any Nios II software. Quartus II adds all HDL files to the archive, including HDL files generated by SOPC Builder, although these files are not strictly necessary.

This chapter is only concerned with archiving the SOPC Builder system, without the generated HDL files, but with all files needed to regenerate them and rebuild the Nios II software (if any).



For more details about archiving Quartus II projects, refer to volume 2 of the *Quartus II Handbook*.

Required Files

This section describes the files required by an SOPC Builder system and its associated Nios II software projects (if any). This is the minimum set of files needed to completely recompile an archived system, both the SRAM Object File (**.sof**) and the executable software (**.elf**).

If you have Nios II software projects, archive them together with the SOPC Builder system on which they are based. You cannot rebuild a Nios II software project without its associated SOPC Builder system.

SOPC Builder Design Files

The files listed in Table 7–1 are located in the Quartus II project directory.

Table 7–1. Files Required for an SOPC Builder System

File description	File name	Write permission required? *(1)*
SOPC Builder system description	<*sopc_builder_system*>**.sopc**	Yes
SOPC Builder legacy system description *(2)*	<*sopc_builder_system*>**.ptf**	Yes
All non-generated HDL source files *(3)*	for example: **top_level_schematic.bdf**, **customlogic.v**	No
Quartus II project file	<*project_name*>**.qpf**	No
Quartus II settings file	<*project_name*>**.qsf**	No

Notes to Table 7–1:
(1) For further information about write permissions, refer to "File Write Permissions" on page 7–4.
(2) The <*sopc_builder_system*>**.ptf** file is only required if you intend to edit or view the system in a version of SOPC Builder prior to version 7.1.
(3) Include all HDL source files not generated by SOPC Builder. This includes HDL source files you create or copy from elsewhere. To identify a file generated by SOPC Builder, open the file and look for the following header:

```
Legal Notice: (C)2007 Altera Corporation. All rights reserved.
```

Nios II Application Software Project Files

The files listed in Table 7–2 are located in the Nios II software project directory.

For more information about Nios II software projects, refer to the *Nios II Software Developer's Handbook*.

Table 7–2. Files Required for a Nios II Application Software Project

File Description	File Name	Write Permission Required? *(1)*
All source files	for example: **app.c**, **header.h**, **assembly.s**, **lookuptable.dat**	No
Eclipse project file	**.project**	No
C/C++ Development Toolkit project file	**.cdtproject**	Yes
C/C++ Development Toolkit option file	**.cdtbuild**	No
Software configuration file	**application.stf**	No

Note to Table 7–2:
(1) For further information about write permissions, refer to "File Write Permissions" on page 7–4.

Nios II System Library Project

The files listed in Table 7–3 are located in the Nios II system library project directory.



For more information about Nios II system libraries, refer to the *Nios II Software Developer's Handbook.*

Table 7–3. Files Required for a Nios II System Library Project

File description	File name	Write permission required? *(1)*
Eclipse project file	**.project**	Yes
C/C++ Development Toolkit project file	**.cdtproject**	Yes
C/C++ Development Toolkit option file	**.cdtbuild**	No
System software configuration file	**system.stf**	Yes

Note to Table 7–3:
(1) For further information about write permissions, see "File Write Permissions" on page 7–4.



Archiving for projects that use Tcl scripting and java to create a Board Support Package (BSP) is covered in chapter 3 of the *Nios II Software Developer's Handbook, Common BSP Tasks*.

File Write Permissions

You must have write permission for certain files. The tools write to these files as part of the generation and compilation process. If the files are not writable, the toolchain fails.

Many source control tools mark local files read-only by default. In this case, you must override this behavior. You do not have to check the files out of source control unless you are modifying the SOPC Builder design or Nios II software project.

Referenced Documents

This chapter references the following documents:

- *The Quartus II Handbook, Volume 2*
- *Nios II Software Developer's Handbook, Common BSP Tasks*

Document Revision History

Table 7–4 shows the revision history for this chapter.

Table 7–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007, v7.2.0	• No change from previous release.	—
May 2007, v7.1.0	• Chapter 7 was previously chapter 6 • Added information about new **.sopc** file type to Table 7–1 • Added information about legacy **.ptf** file type to Table 7–1 • Added Referenced Documents section • Added reference to new Common BSP Tasks chapter for archiving of Tcl projects	Updates to this chapter include replacing the legacy **.ptf** file type with the new **.sopc** file type.
March 2007, v7.0.0	• No change from previous release	—
November 2007, v6.1.0	• No change from previous release	—
May 2006, v6.0.0	Initial release.	—

Section II. Building Systems with SOPC Builder

This section provides instructions on how to use SOPC Builder to achieve specific goals. Chapters in this section serve to answer the question, "How do I use SOPC Builder?" Many chapters in this handbook provide design examples that you can download free from **www.altera.com**. Design file hyperlinks are located with individual chapters linked from the Altera web site.

This section includes the following chapters:

- Chapter 8, Building Memory Subsystems Using SOPC Builder
- Chapter 9, Developing Components for SOPC Builder



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

8. Building Memory Subsystems Using SOPC Builder

QII54006-7.2.0

Introduction

Most systems generated with SOPC Builder require memory. For example, embedded processor systems require memory for software code, while digital signal processing (DSP) systems require memory for data buffers. Many systems use multiple types of memories. For example, a processor-based DSP system can use off-chip SDRAM to store software code, and on-chip RAM for fast access to data buffers. You can use SOPC Builder to integrate almost any type of memory into your system.

This chapter describes how to build a memory subsystem as part of a larger system created with SOPC Builder. This chapter focuses on the following kinds of memory most commonly used in SOPC Builder systems for:

- "On-Chip RAM and ROM" on page 8–8
- "EPCS Serial Configuration Device" on page 8–12
- "SDRAM" on page 8–14
- "Off-Chip SRAM and Flash Memory" on page 8–19

This chapter assumes that you are familiar with the following:

- Creating FPGA designs and making pin assignments with the Quartus® II software. For details, refer to the *Introduction to the Quartus II Software* manual.
- Building simple systems with SOPC Builder. For details, refer to the *Introduction to SOPC Builder* in volume 4 of the *Quartus II Handbook*.
- SOPC Builder components. For details, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon® interfaces. You do not need extensive knowledge of the Avalon interfaces, such as transfer types or signal timing. However, to create your own custom memory subsystem with external memories, you need to understand the Avalon® Memory-Mapped (Avalon-MM) interface. For details, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces* chapter in volume 4 of the *Quartus II Handbook* and the *Avalon Memory-Mapped Interface Specification*.

Example Design

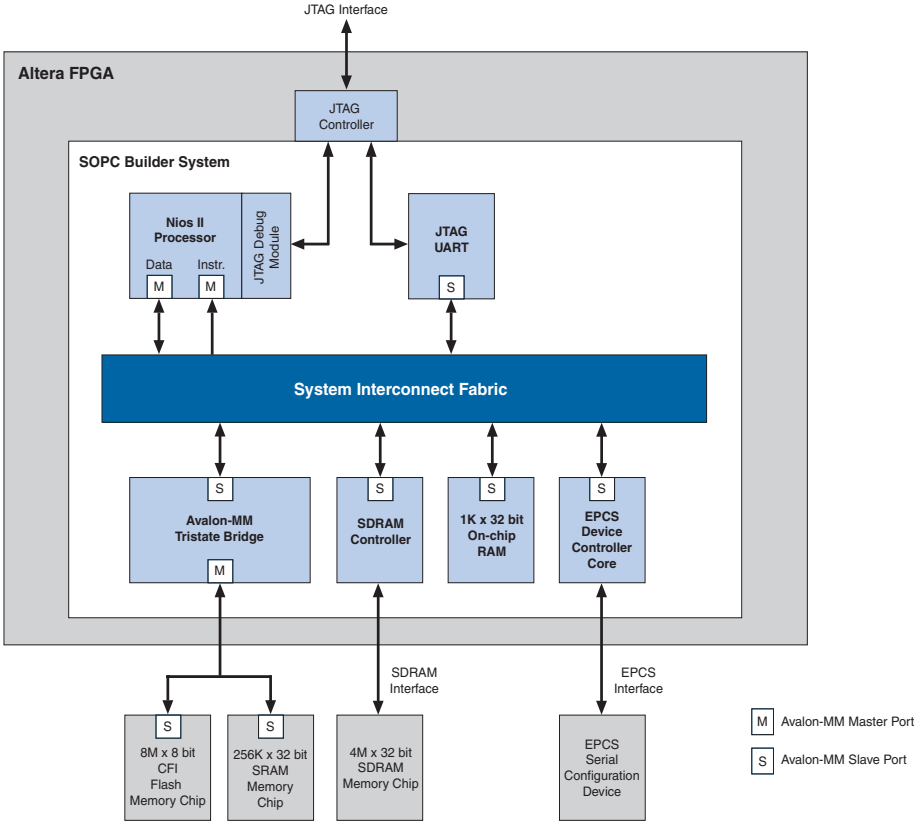
This chapter demonstrates the process for building a system that contains one of each type memory as shown in Figure 8–1. Each section of the chapter builds on previous sections, culminating in a complete system.

By following the example design in this chapter, you will learn how to create a complete customized memory subsystem for your system or design. The memory components in the example design are independent. For a custom system, you can instantiate exactly the memories you need, and skip the memories you do not need. Furthermore, you can create multiple instantiations of the same type of memory, limited only by on-chip memory resources or FPGA pins to interface with off-chip memory devices.

Example Design Structure

Figure 8–1 shows a block diagram of the example system.

Figure 8–1. Example Design Block Diagram



In Figure 8–1, all blocks shown below the system interconnect fabric comprise the memory subsystem. For demonstration purposes, this system uses a Nios® II processor core to master the memory devices, and a JTAG UART core to communicate with the processor. However, the memory subsystem could be connected to any master component, either on-chip or off-chip.

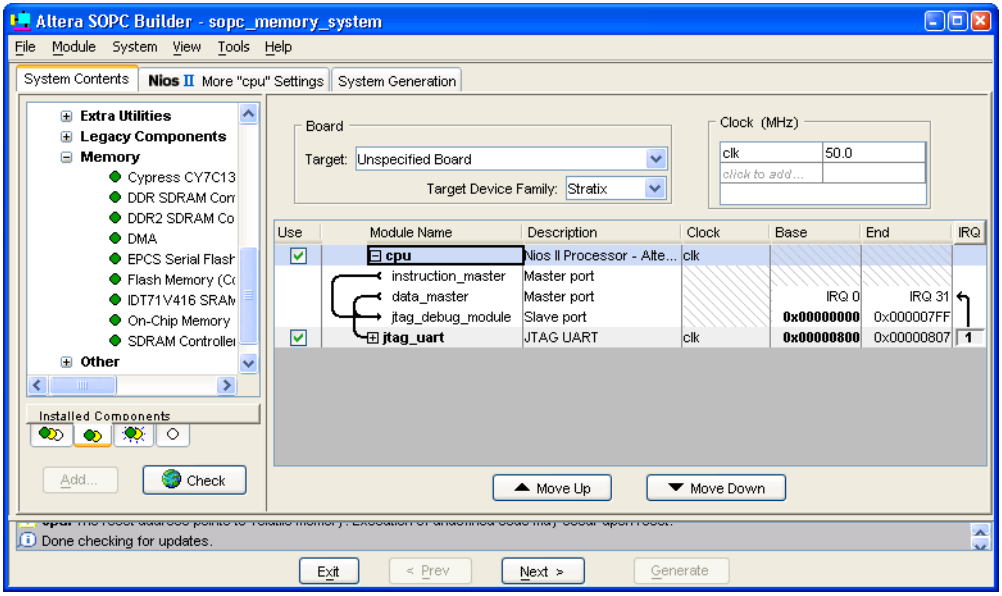
Example Design Starting Point

The example design consists of the following elements:

- A Quartus II project named **quartus2_project**. A Block Design File (**.bdf**) named **toplevel_design**. **toplevel_design** is the top-level design file for **quartus2_project**. **toplevel_design** instantiates the SOPC Builder system module, as well as other pins and modules required to complete the design.
- An SOPC Builder system named **sopc_memory_system**. **sopc_memory_system** is a subdesign of **toplevel_design**. **sopc_memory_system** instantiates the memory components and other SOPC Builder components required for a functioning system module.

The starting point for this chapter assumes that the **quartus2_project** already exists, **sopc_memory_system** has been started in SOPC Builder, and the Nios II core and the JTAG UART core are already instantiated. This example design uses the default settings for the Nios II/s core and the JTAG UART core; these settings do not affect the rest of the memory subsystem. Figure 8–2 shows the starting point in the SOPC Builder.

Figure 8–2. Starting Point for the Example Design



All sections in this chapter build on this starting point.

Hardware and Software Requirements

To build a memory subsystem similar to the example design in this chapter, you need the following:

- Quartus II Software version 5.0 or higher—Both Quartus II Web Edition and the fully licensed version support this design flow.
- Nios II Embedded Design Suite (EDS) version 5.0 or higher—Both the evaluation edition and the fully licensed version support this design flow. The Nios II EDS provides the SOPC Builder memory components described in this chapter. It also provides several complete example designs which demonstrate a variety of memory components instantiated in working systems.



The Quartus II Web Edition software and the Nios II EDS, Evaluation Edition are available free for download from the Altera® website. Visit **www.altera.com/download**.

This chapter does not describe downloading and verifying a working system in hardware. Therefore, there are no hardware requirements for the completion of this chapter. However, the example memory subsystem has been tested in hardware.

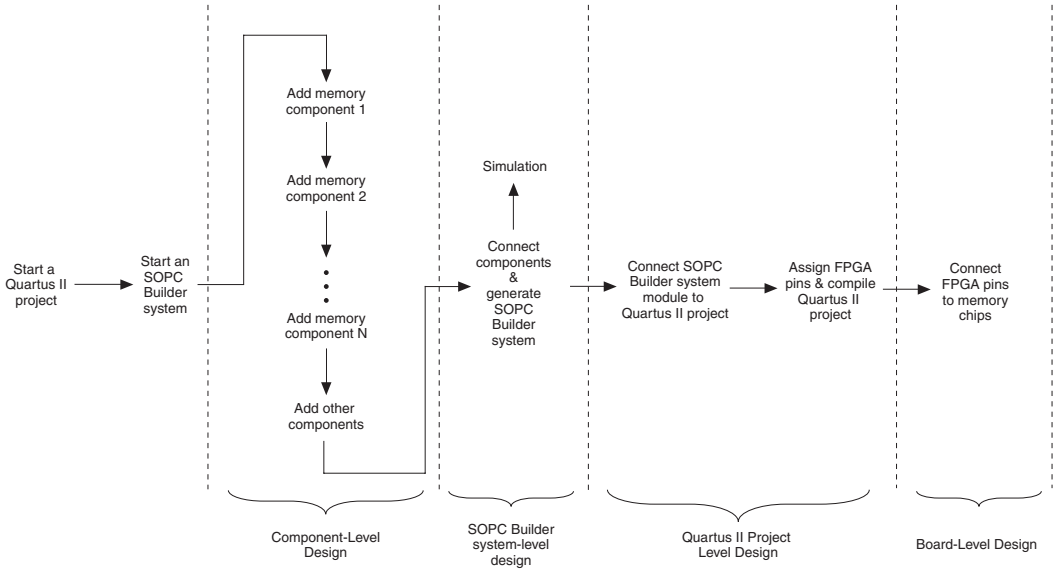
Design Flow

This section describes the design flow for building memory subsystems with SOPC Builder.

The design flow for building a memory subsystem is similar to other SOPC Builder designs. After starting a Quartus II project and an SOPC Builder system, there are five steps to completing the system, as shown in Figure 8–3:

1. Component-level design in SOPC Builder
2. SOPC Builder system-level design
3. Simulation
4. Quartus II project-level design
5. Board-level design

Figure 8–3. Design Flow



Component-Level Design in SOPC Builder

In this step, you specify which memory components to use and configure each component to meet the needs of the system. All memory components are available from the **Memory and Memory Controllers** category in the SOPC Builder list of available components.

SOPC Builder System-Level Design

In this step, you connect components together and configure the SOPC Builder system as a whole. Similar to the process of adding non-memory SOPC Builder components, you use the SOPC Builder **System Contents** tab to do the following:

- Rename the component instance (optional).
- Connect the memory component to master ports in the system. Each memory component must be connected to at least one master port.
- Assign a base address.
- Assign a clock domain. A memory component can operate on the same or different clock domain as the master port(s) that access it.

Simulation

In this step, you verify the functionality of the SOPC Builder system module. For systems with memories, this step depends on simulation models for each of the memory components, in addition to the system test bench generated by SOPC Builder. Refer to "Simulation Considerations" for more information.

Quartus II Project-Level Design

In this step, you integrate the SOPC Builder system module with the rest of the Quartus II project. This step includes wiring the system module to FPGA pins, and wiring the system module to other design blocks (such as other HDL modules) in the Quartus II project.



In the example design in this chapter, the SOPC Builder system module comprises the entire FPGA design. There are no other design blocks in the Quartus II project.

Board-Level Design

In this step, you connect the physical FPGA pins to memory devices on the board. If the SOPC Builder system interfaces with off-chip memory devices, you must make board-level design choices.

Simulation Considerations

SOPC Builder can automatically generate a test bench for register transfer level (RTL) simulation of the system. This test bench instantiates the system module and can also instantiate memory models for external memory components. The test bench is plain text HDL, located at the bottom of the top-level system module HDL design file. To explore the contents of the auto-generated test bench, open the top-level HDL file and search on keyword `test_bench`.

Generic Memory Models

The memory components described in this chapter, except for the SRAM, provide generic simulation models. Therefore, it is very easy to simulate an SOPC Builder system with memory components immediately after generating the system.

The generic memory models store memory initialization files, such as Data [file name extension] (**.dat**) and Hexadecimal (**.hex**) files, in a directory named *<Quartus II project directory>/<SOPC Builder system*

name>_**sim**. When generating a new system, SOPC Builder creates empty initialization files. You can manually edit these files to provide custom memory initialization contents for simulation.



For Nios II processor designs, the Nios II integrated development environment (IDE) generates initialization contents automatically.

Vendor-Specific Memory Models

You can also manually connect vendor-specific memory models to the system module. In this case, you must manually edit the testbench and connect the vendor memory model. You might also need to edit the vendor memory model slightly for time delays. The SOPC Builder testbench assumes zero delay.

On-Chip RAM and ROM

Altera FPGAs include on-chip memory blocks that can be used as RAM or ROM in SOPC Builder systems. On-chip memory has the following benefits for SOPC Builder systems:

- On-chip memory has fast access time, compared to off-chip memory.
- SOPC Builder automatically instantiates on-chip memory inside the system module, so you do not have to make any manual connections.
- Certain memory blocks can have initialized contents when the FPGA powers up. This feature is useful, for example, for storing data constants or processor boot code.

FPGAs have limited on-chip memory resources, which limits the maximum practical size of an on-chip memory to approximately one megabyte in the largest FPGA family.

Component-Level Design for On-Chip Memory

In SOPC Builder you instantiate on-chip memory by clicking the **On-chip Memory (RAM or ROM)** in the component. The configuration wizard for the **On-chip Memory (RAM or ROM)** component has the following options: **Memory Type**, **Size**, and **Read Latency**.

Memory Type

The **Memory Type** options define the structure of the on-chip memory:

- **RAM (writable)**—This setting creates a readable and writable memory.
- **ROM (read only)**—This setting creates a read-only memory.

- **Dual-port access**—Turning on this setting creates a memory component with two slave ports, which allows two master ports to access the memory simultaneously.
- **Block type**—This setting directs the Quartus II software to use a specific type of memory block when fitting the on-chip memory in the FPGA. The following choices are available:
  - **Auto**—This setting allows the Quartus II software to choose the most appropriate memory resource.
  - **M512**—This setting directs the Quartus II software to use M512 blocks.
  - **M4K**—This setting directs the Quartus II software to use M4K blocks.
  - **M-RAM**—This setting directs the Quartus II software to use M-RAM blocks. The 64 Kbit M-RAM blocks are appropriate for larger RAM data buffers. However, M-RAM blocks do not allow pre-initialized contents at power up.

Size

The **Size** options define the size and width of the memory.

- **Data width**—This setting determines the data width of the memory. The available choices are **8**, **16**, **32**, **64**, **128**, **256**, **512**, or **1024** bits. Assign **Data width** to match the width of the master port that accesses this memory the most frequently or has the most critical timing requirements.
- **Total memory size**—This setting determines the total size of the on-chip memory block. The total memory size must be less than the available memory in the target FPGA.

Read Latency

On-chip memory components use synchronous, pipelined Avalon-MM slave ports. Pipelined access improves f_{MAX} performance, but also adds latency cycles when reading the memory. The **Read latency** option allows you to specify the number of read latency cycles required to access data. If the **Dual-port access** setting is turned on, you can specify a different read latency for each slave port.

Non-Default Memory Initialization

For ROM memories, you can specify your own initialization file by selecting **Enable non-default initialization file**. If this option is selected, the file you specify will be used to initialize the ROM in place of the default initialization file created by SOPC Builder.

Enable In-System Memory Content Editor Feature

Allows you to enable the In-System Memory Content Editor, which allows you to read data from and write data to in-system memory in a device while the device is running at speed and independently of system clocks with a JTAG interface.

SOPC Builder System-Level Design for On-Chip Memory

There are few SOPC Builder system-level design considerations for on-chip memories. See "SOPC Builder System-Level Design" on page 8–6.

When generating a new system, SOPC Builder creates a blank initialization file in the Quartus II project directory for each on-chip memory that can power up with initialized contents. The name of this file is <*name of memory component*>**.hex**.

Simulation for On-Chip Memory

At system generation time, SOPC Builder generates a simulation model for the on-chip memory. This model is embedded inside the system module, and there are no user-configurable options for the simulation testbench.

You can provide memory initialization contents for simulation in the file <*Quartus II project directory*>/<*SOPC Builder system name*>**_sim/**<*Memory component name*>**.dat**.

Quartus II Project-Level Design for On-Chip Memory

The on-chip memory is embedded inside the SOPC Builder system module, and therefore there are no signals to connect to the Quartus II project.

To provide memory initialization contents, you must fill in the file <*name of memory component*>**.hex.** The Quartus II software recognizes this file during design compilation and incorporates the contents into the configuration files for the FPGA.



For Nios II processor users, the Nios II integrated development environment (IDE) generates the memory initialization file automatically.

Board-Level Design for On-Chip Memory

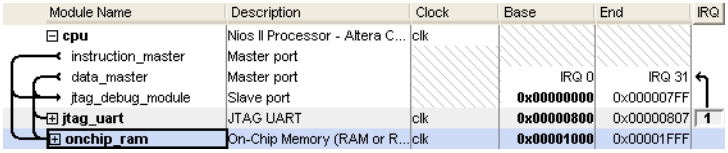
The on-chip memory is embedded inside the SOPC Builder system module, and therefore there is nothing to connect at the board level.

Example Design with On-Chip Memory

This section demonstrates adding a 4 Kbyte on-chip RAM to the example design. This memory uses a single slave port with read latency of one cycle.

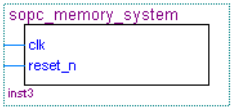
Figure 8–4 shows the SOPC Builder system after adding an instance of the on-chip memory component, renaming it to `onchip_ram`, and assigning it a base address.

Figure 8–4. SOPC Builder System with On-Chip Memory



For demonstration purposes, Figure 8–5 shows the result of generating the system module at this stage. (In a normal design flow, you generate the system only after adding all system components.)

Figure 8–5. System Module with On-Chip Memory



Because the on-chip memory is contained entirely within the system module, **sopc_memory_system** has no I/O signals associated with **onchip_ram**. Therefore, you do not need to make any Quartus II project connections or assignments for the on-chip RAM, and there are no board-level considerations.

EPCS Serial Configuration Device

Many systems use an Altera EPCS serial configuration device to configure the FPGA. Altera provides the EPCS device controller core, which allows SOPC Builder systems to access the memory contents of the EPCS device. This feature provides flexible design options:

- The FPGA design can reprogram its own configuration memory, providing a mechanism for in-field upgrades.
- The FPGA design can use leftover space in the EPCS as nonvolatile storage.

Physically, the EPCS device is a serial flash memory device, which has slow access time. Altera provides software drivers to control the EPCS core for the Nios II processor only. Therefore, EPCS controller core features are available only to SOPC Builder systems that include a Nios II processor.



For further details about the features and usage of the EPCS device controller core, refer to the *EPCS Device Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

Component-Level Design for an EPCS Device

In SOPC Builder you instantiate an EPCS controller core by adding an **EPCS Serial Flash Controller** component. There are no settings for this component.



For details, refer to the *Nios II Flash Programmer User Guide*.

SOPC Builder System-Level Design for an EPCS Device

There are not many SOPC Builder system-level design considerations for EPCS devices:

- Assign a base address.
- Set the IRQ connection to **NC** (disconnected). The EPCS controller hardware is capable of generating an IRQ. However, the Nios II driver software does not use this IRQ, and therefore you can leave the IRQ signal disconnected.

There can only be one EPCS controller core per FPGA, and the instance of the core is always named `epcs_controller`.

Simulation for an EPCS Device

The EPCS controller core provides a limited simulation model:

- Functional simulation does not include the FPGA configuration process, and therefore the EPCS controller does not model the configuration features.
- The simulation model does not support read and write operations to the flash region of the EPCS device.
- A Nios II processor can boot from the EPCS device in simulation. However, the boot loader code is different during simulation. The EPCS controller boot loader code assumes that all other memory simulation models are pre-initialized, and therefore the boot load process is unnecessary. During simulation, the boot loader simply forces the Nios II processor to jump to start, skipping the boot load process.

Verification in the hardware is the best way to test features related to the EPCS device.

Quartus II Project-Level Design for an EPCS Device

The Quartus II software automatically connects the EPCS controller core in the SOPC Builder system to the dedicated configuration pins on the FPGA. This connection is invisible to the user. Therefore, there are no EPCS-related signals to connect in the Quartus II project.

Board-Level Design for an EPCS Device

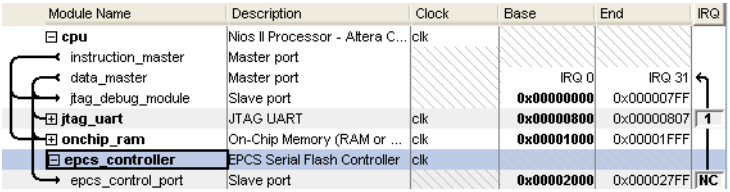
You must connect the EPCS device to the FPGA as described in the Altera *Configuration Handbook*. No other connections are necessary.

Example Design with an EPCS Device

This section demonstrates adding an EPCS device controller core to the example design.

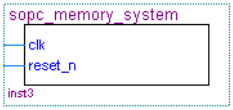
Figure 8–6 shows the SOPC Builder system after adding an instance of the EPCS controller core and assigning it a base address.

Figure 8–6. SOPC Builder System with EPCS Device



For demonstration purposes only, Figure 8–7 shows the result of generating the system module at this stage.

Figure 8–7. System Module with EPCS Device



Because the Quartus II software automatically connects the EPCS controller core to the FPGA pins, the system module has no I/O signals associated with **epcs_controller**. Therefore, you do not need to make any Quartus II project connections or assignments for the EPCS controller core.

This chapter does not cover the details of configuration using the EPCS device. For further information, refer to Altera's *Configuration Handbook*.

SDRAM

Altera provides a free SDRAM controller core, which allows you to use inexpensive SDRAM as bulk RAM in your FPGA designs. The SDRAM controller core is necessary, because Avalon-MM signals cannot describe the complex interface on an SDRAM device. The SDRAM controller acts as a bridge between the system interconnect fabric and the pins on an SDRAM device. The SDRAM controller can operate in excess of 100 MHz.

For further details about the features and usage of the SDRAM controller core, refer to the *SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

Component-Level Design for SDRAM

The choice of SDRAM device(s) and the configuration of the device(s) on the board heavily influence the component-level design for the SDRAM controller. Typically, the component-level design task involves parameterizing the SDRAM controller core to match the SDRAM device(s) on the board. You must specify the structure (address width, data width, number of devices, number of banks, and so on) and the timing specifications of the device(s) on the board.



For complete details about configuration options for the SDRAM controller core, refer to the *SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

SOPC Builder System-Level Design for SDRAM

In the SOPC Builder **System Contents** tab, the SDRAM controller looks like any other memory component. Similar to on-chip memory, there are few SOPC Builder system-level design considerations for SDRAM. See "SOPC Builder System-Level Design" on page 8–6.

Simulation for SDRAM

At system generation time, SOPC Builder can generate a generic SDRAM simulation model and include the model in the system testbench. To use the generic SDRAM simulation model, you must turn on a setting in the SDRAM controller configuration wizard. You can provide memory initialization contents for simulation in the file <*Quartus II project directory*>/<*SOPC Builder system name*>**_sim/**<*Memory component name*>**.dat.**

Alternately, you can provide a specific vendor memory model for the SDRAM. In this case, you must manually wire up the vendor memory model in the system testbench.



For further details, refer to "Simulation Considerations" on page 8–7 and the *SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

Quartus II Project-Level Design for SDRAM

SOPC Builder generates a system module with top-level I/O signals associated with the SDRAM controller. In the Quartus II project, you must connect these I/O signals to FPGA pins, which connect to the SDRAM device on the board. In addition, you might have to accommodate clock skew issues.

Connecting and Assigning the SDRAM-Related Pins

After generating the system with SOPC Builder, you can find the names and directions of the I/O signals in the top-level HDL file for the SOPC Builder system module. The file has the name <*Quartus II project directory*>/<*SOPC Builder system name*>**.v** or <*Quartus II project directory*>/<*SOPC Builder system name*>**.vhd**. You must connect these signals in the top-level Quartus II design file.

You must assign a pin location for each I/O signal in the top-level Quartus II design to match the target board. Depending on the performance requirements for the design, you might have to assign FPGA pins carefully to achieve performance.

Accommodating Clock Skew

As SDRAM frequency increases, so does the possibility that you must accommodate skew between the SDRAM clock and I/O signals. This issue affects all synchronous memory devices, including SDRAM. To accommodate clock skew, you can instantiate an altpll megafunction in the top-level Quartus II design to create a phase-locked loop (PLL) clock output. You use a phase-shifted PLL output to drive the SDRAM clock and reduce clock-skew issues. The exact settings for the altpll megafunction depend on your target hardware; you must experiment to tune the phase shift to match the board.

For details, refer to the *altpll Megafunction User Guide*.

Board-Level Design for SDRAM

Memory requirements largely dictate the board-level configuration of the SDRAM device(s). The SDRAM controller core can accommodate various configurations of SDRAM on the board, including multiple banks and multiple devices.

For further details, refer to the *SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

Example Design with SDRAM

This section demonstrates adding a 16-Mbyte SDRAM device to the example design. This SDRAM is a single device with 32-bit data. Figure 8–8 shows the **SDRAM Controller** configuration wizard settings for the example design.

Figure 8–8. SDRAM Controller Configuration Wizard

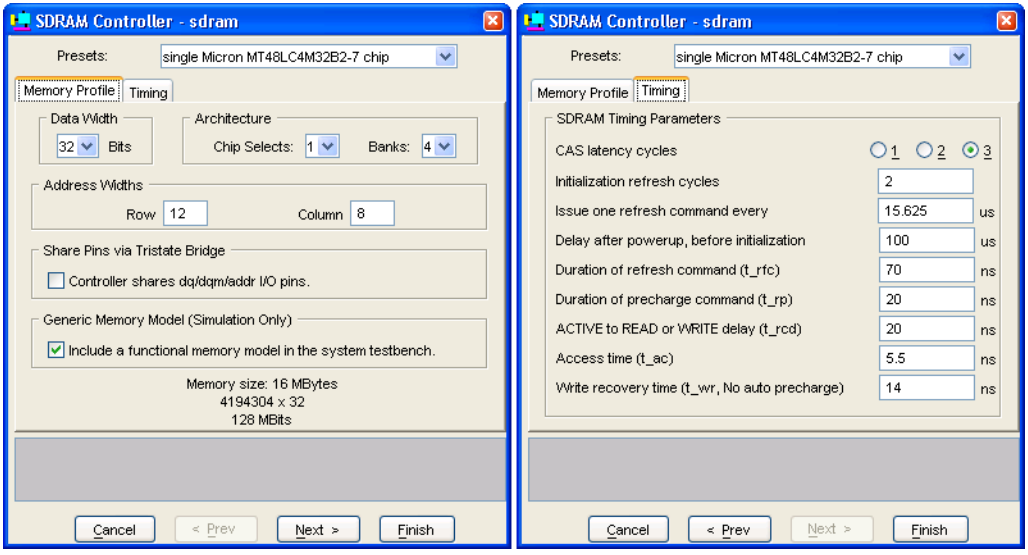
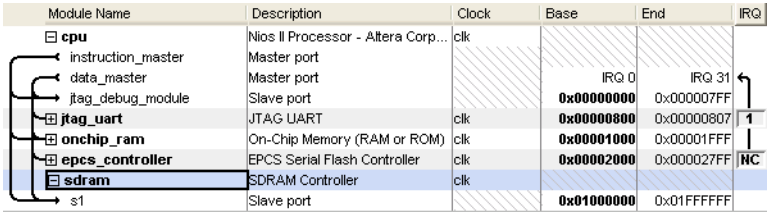


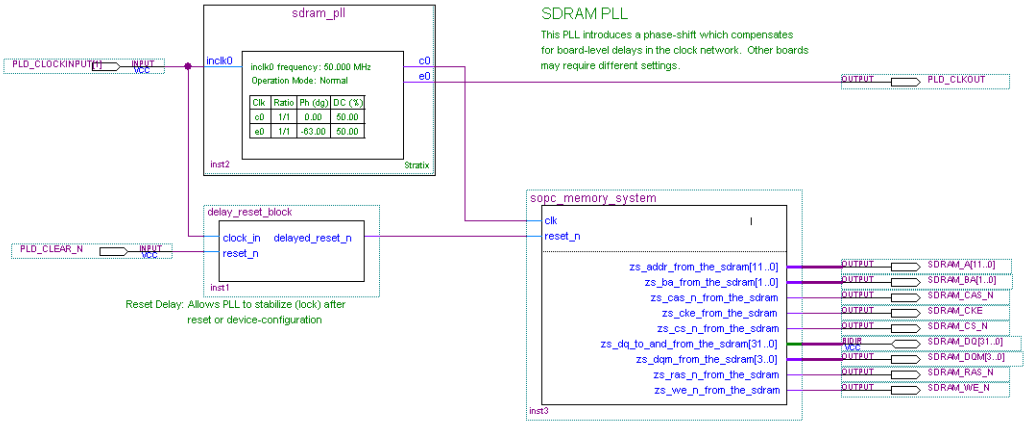
Figure 8–9 shows the SOPC Builder system after adding an instance of the SDRAM controller, renaming it to **sdram**, and assigning it a base address.

Figure 8–9. SOPC Builder System with SDRAM



For demonstration purposes, Figure 8–10 shows the result of generating the system module at this stage, and connecting it in **toplevel_design.bdf**.

Figure 8–10. toplevel_design.bdf with SDRAM



After generating the system, the top-level system module file **sopc_memory_system.v** contains the list of SDRAM-related I/O signals which must be connected to FPGA pins:

```
output   [ 11: 0] zs_addr_from_the_sdram;
output   [  1: 0] zs_ba_from_the_sdram;
output            zs_cas_n_from_the_sdram;
output            zs_cke_from_the_sdram;
output            zs_cs_n_from_the_sdram;
inout    [ 31: 0] zs_dq_to_and_from_the_sdram;
output   [  3: 0] zs_dqm_from_the_sdram;
output            zs_ras_n_from_the_sdram;
output            zs_we_n_from_the_sdram;
```

As shown in Figure 8–10, **toplevel_design.bdf** uses an instance of `sdram_pll` to phase shift the SDRAM clock by –63 degrees. **toplevel_design.bdf** also uses a subdesign `delay_reset_block` to insert a delay on the `reset_n` signal for the system module. This delay is necessary to allow the PLL output to stabilize before the SOPC Builder system begins operating.

Figure 8–11 shows pin assignments in the Quartus II Assignment Editor for some of the SDRAM pins. The correct pin assignments depend on the target board.

Figure 8–11. Pin Assignments for SDRAM

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reserved
188	SDRAM_A[0]	PIN_AE4	7	LVTTL	Column I/O		
189	SDRAM_A[10]	PIN_Y11	7	LVTTL	Column I/O		
190	SDRAM_A[11]	PIN_AB7	7	LVTTL	Column I/O		
191	SDRAM_A[1]	PIN_W12	7	LVTTL	Column I/O	PGM0	
192	SDRAM_A[2]	PIN_AC11	7	LVTTL	Column I/O	nRS	
193	SDRAM_A[3]	PIN_W10	7	LVTTL	Column I/O	RUnLU	
194	SDRAM_A[4]	PIN_AA11	7	LVTTL	Column I/O	PGM1	
195	SDRAM_A[5]	PIN_AC10	7	LVTTL	Column I/O	RDN7	
196	SDRAM_A[6]	PIN_AB11	7	LVTTL	Column I/O	RUP7	
197	SDRAM_A[7]	PIN_AC8	7	LVTTL	Column I/O	FCLK5	
198	SDRAM_A[8]	PIN_AB10	7	LVTTL	Column I/O	FCLK4	
199	SDRAM_A[9]	PIN_V11	7	LVTTL	Column I/O		
200	SDRAM_BA[0]	PIN_AG19	8	LVTTL	Column I/O	DQ6B4	
201	SDRAM_BA[1]	PIN_AF19	8	LVTTL	Column I/O	DQ6B5	
202	SDRAM_CAS_N	PIN_AD18	8	LVTTL	Column I/O	DQ6B2	
203	SDRAM_CKE	PIN_AE18	8	LVTTL	Column I/O	DQ6B1	
204	SDRAM_CS_N	PIN_AG18	8	LVTTL	Column I/O	DQ6B0	
205	SDRAM_DQM[0]	PIN_AE14	7	LVTTL	Column I/O	CLK6n	
206	SDRAM_DQM[1]	PIN_Y13	7	LVTTL	Column I/O	CLK7n	
207	SDRAM_DQM[2]	PIN_AE7	7	LVTTL	Column I/O	DQS1B	
208	SDRAM_DQM[3]	PIN_AG10	7	LVTTL	Column I/O	DQS3B	

Off-Chip SRAM and Flash Memory

SOPC Builder systems can directly access many off-chip RAM and ROM devices, without a controller core to drive the off-chip memory. Avalon-MM signals can exactly describe the interfaces on many standard memories, such as SRAM and flash memory. In this case, I/O signals on the SOPC Builder system module can connect directly to the memory device.

While off-chip memory usually has slower access time than on-chip memory, off-chip memory provides the following benefits:

- Off-chip memory is less expensive than on-chip memory resources.
- The size of off-chip memory is bounded only by the 32-bit Avalon-MM address space.
- Off-chip ROM, such as flash memory, can be used for bulk storage of nonvolatile data.
- Multiple off-chip RAM and ROM memories can share address and data pins to conserve FPGA I/O resources.

Adding off-chip memories to an SOPC Builder system also requires the **Avalon-MM Tristate Bridge** component.

This section describes the process of adding off-chip flash memory and SRAM to an SOPC Builder system.

Component-Level Design for SRAM and Flash Memory

There are several ways to instantiate an interface to an off-chip memory device:

- For common flash interface (CFI) flash memory devices, add the **Flash Memory (Common Flash Interface)** component in SOPC Builder.
- For Altera development boards, Altera provides SOPC Builder components that interface to the specific devices on each development board. For example, the Nios II EDS includes the components **Cypress CY7C1380C SSRAM** and **IDT71V416 SRAM**, which appear on Nios II development boards.
- For further details about the features and usage of the SSRAM controller core, refer to the *SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.
- For further details about the features and usage of the SDRAM controller core, refer to the *Building Memory Subsystems Using SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.

These components make it easy for you to create memory systems targeting Altera development boards. However, these components target only the specific memory device on the board; they do not work for different devices.

- For general memory devices, RAM or ROM, you can create a custom interface to the device with the SOPC Builder component editor. Using the component editor, you define the I/O pins on the memory device and the timing requirements of the pins.

In all cases, you must also instantiate the **Avalon-MM Tristate Bridge** component. Multiple off-chip memories can connect to a single tristate bridge.

Avalon-MM Tristate Bridge

A tristate bridge connects off-chip devices to on-chip system interconnect fabric. The tristate bridge creates I/O signals on the SOPC Builder system module, which you must connect to FPGA pins in the top-level Quartus II project. These pins represent the system interconnect fabric to off-chip devices.

The tristate bridge creates address and data pins which can be shared by multiple off-chip devices. This feature lets you conserve FPGA pins when connecting the FPGA to multiple devices with mutually exclusive access.

You must use a tristate bridge in either of the following cases:

- The off-chip device has bidirectional data pins.
- Multiple off-chip devices share the address and/or data buses.

In SOPC Builder, you instantiate a tristate bridge by instantiating the **Avalon-MM Tristate Bridge** component. The Avalon-MM Tristate Bridge configuration wizard has a single option: To register incoming (to the FPGA) signals or not.

- **Registered**—This setting adds registers to all FPGA input pins associated with the tristate bridge (outputs from the memory device).
- **Not Registered**—This setting does not add registers between the memory device output pins and the system interconnect fabric.

The Avalon-MM tristate bridge automatically adds registers to output signals from the tristate bridge to off-chip devices.

Registering the input and output signals shortens the register-to-register delay from the memory device to the FPGA, resulting in higher system f_{MAX} performance. However, in each direction, the registers add one additional cycle of latency for Avalon-MM master ports accessing memory connected to the tristate bridge. The registers do not affect the timing of the transfers from the perspective of the memory device.



For details about the Avalon-MM tristate interface, refer to the *Avalon Memory-Mapped Interface Specification*.

Flash Memory

In SOPC Builder, you instantiate an interface to CFI flash memory by adding a **Flash Memory (Common Flash Interface)** component. If the flash memory is not CFI compliant, you must create a custom interface to the device with the SOPC Builder component editor.

The choice of flash device(s) and the configuration of the device(s) on the board heavily influence the component-level design for the flash memory configuration wizard. Typically, the component-level design task involves parameterizing the flash memory interface to match the device(s) on the board. Using the Flash Memory (Common Flash Interface) configuration wizard, you must specify the structure (address width and data width) and the timing specifications of the device(s) on the board.



For details about features and usage, refer to the *Common Flash Interface Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

For an example of instantiating the Flash Memory (Common Flash Interface) component in an SOPC Builder system, see "Example Design with SRAM and Flash Memory" on page 8–25.

SRAM

To instantiate an interface to off-chip RAM, perform the following steps:

1. Create a new component with the SOPC Builder component editor that defines the interface.
2. Instantiate the new interface component in the SOPC Builder system.

The choice of RAM device(s) and the configuration of the device(s) on the board determine how you create the interface component. The component-level design task involves entering parameters into the component editor to match the device(s) on the board.



For details about using the component editor, refer to the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*.

SOPC Builder System-Level Design for SRAM and Flash Memory

In the SOPC Builder **System Contents** tab, the Avalon-MM tristate bridge has two ports:

- Avalon-MM slave port—This port faces the on-chip logic in the SOPC Builder system. You connect this slave port to on-chip master ports in the system.
- Avalon-MM tristate master port—This port faces the off-chip memory devices. You connect this master port to the Avalon-MM tristate slave ports on the interface components for off-chip memories.

You assign a clock to the Avalon-MM tristate bridge that determines the Avalon-MM clock cycle time for off-chip devices connected to the tristate bridge.

You must assign base addresses to each off-chip memory. The Avalon-MM tristate bridge does not have an address; it passes unmodified addresses from on-chip master ports to off-chip slave ports.

Simulation for SRAM and Flash Memory

The SOPC Builder output for simulation depends on the type of memory component(s) in the system:

- **Flash Memory (Common Flash Interface)** component—This component provides a generic simulation model. You can provide memory initialization contents for simulation in the file <*Quartus II project directory*>/<*SOPC Builder system name*>**_sim/**<*Flash memory component name*>**.dat**.
- Custom memory interface created with the component editor—In this case, you must manually connect the vendor simulation model to the system test bench. SOPC Builder does not automatically connect simulation models for custom memory components to the system module.
- Altera-provided interfaces to memory devices—Altera provides simulation models for these interface components. You can provide memory initialization contents for simulation in the file <*Quartus II project directory*>/<*SOPC Builder system name*>**_sim/**<*Memory component name*>**.dat**. Alternately, you can provide a specific vendor simulation model for the memory. In this case, you must manually wire up the vendor memory model in the system test bench.

For further details, see "Simulation Considerations" on page 8–7.

Quartus II Project-Level Design for SRAM and Flash Memory

SOPC Builder generates a system module with top-level I/O signals associated with the tristate bridge and the memory interface components. In the Quartus II project, you must connect the I/O signals to FPGA pins, which connect to the memory device(s) on the board.

After generating the system with SOPC Builder, you can find the names and directions of the I/O signals in the top-level HDL file for the SOPC Builder system module. The file has the name <*Quartus II project directory*>/<*SOPC Builder system name*>**.v** or <*Quartus II project directory*>/<*SOPC Builder system name*>**.vhd**. You must connect these signals in the top-level Quartus II design file.

You must assign a pin location for each I/O signal in the top-level Quartus II design to match the target board. Depending on the performance requirements for the design, you might have to assign FPGA pins carefully to achieve performance.

SOPC Builder inserts synthesis directives in the top-level system module HDL to assist the Quartus II fitter with signals that interface with off-chip devices. The following is an example:

```
reg [ 22: 0] tri_state_bridge_address /* synthesis
ALTERA_ATTRIBUTE =  "FAST_OUTPUT_REGISTER=ON" */;
```

Board-Level Design for SRAM and Flash Memory

Memory requirements largely dictate the board-level configuration of the SRAM and flash memory device(s). You can lay out memory devices in any configuration, as long as the resulting interface can be described with Avalon-MM signals.



Special consideration is required when connecting the Avalon-MM `address` signal to the address pins on the memory devices.

The system module presents the smallest number of address lines required to access the largest off-chip memory, which is usually less than 32 address bits. Not all memory devices connect to all address lines.

Aligning the Least-Significant Address Bits

The Avalon-MM tristate `address` signal always presents a byte address. Each address location in many memory devices contains more than one byte of data. In this case, the memory device must ignore one or more of the least-significant Avalon-MM `address` lines. For example, a 16-bit memory device must ignore Avalon-MM `address[0]` (which is a byte address), and connect Avalon-MM `address[1]` to the least-significant address line.

Table 8–1 shows the relationship between Avalon-MM `address` lines and off-chip address pins for all possible Avalon-MM data widths.

Table 8–1. Connecting the Least-Significant Avalon-MM Address Line

Avalon-MM Address Line	Address Line on Memory Device				
	8-bit Memory	16-bit Memory	32-bit Memory	64-bit Memory	128-bit Memory
`address[0]`	A0	No connect	No connect	No connect	No connect
`address[1]`	A1	A0	No connect	No connect	No connect
`address[2]`	A2	A1	A0	No connect	No connect
`address[3]`	A3	A2	A1	A0	No connect
`address[4]`	A4	A3	A2	A1	A0
`address[5]`	A5	A4	A3	A2	A1
`address[6]`	A6	A5	A4	A3	A2
`address[7]`	A7	A6	A5	A4	A3
`address[8]`	A8	A7	A6	A5	A4
`address[9]`	A9	A8	A7	A6	A5
`address[10]`	A10	A9	A8	A7	A6
`...`	...	...	...	...	...

Aligning the Most-Significant Address Bits

The Avalon-MM `address` signal contains enough address lines for the largest memory on the tristate bridge. Smaller off-chip memories might not use all of the most-significant address lines.

For example, a memory device with 2^{10} locations uses 10 address bits, while a memory with 2^{20} locations uses 20 address bits. If both these devices share the same tristate bridge, the smaller memory ignores the ten most significant Avalon-MM address lines.

Example Design with SRAM and Flash Memory

This section demonstrates adding a 1-Mbyte SRAM and an 8-Mbyte flash memory to the example design. These memory devices connect to the system interconnect fabric through an Avalon-MM tristate bridge.

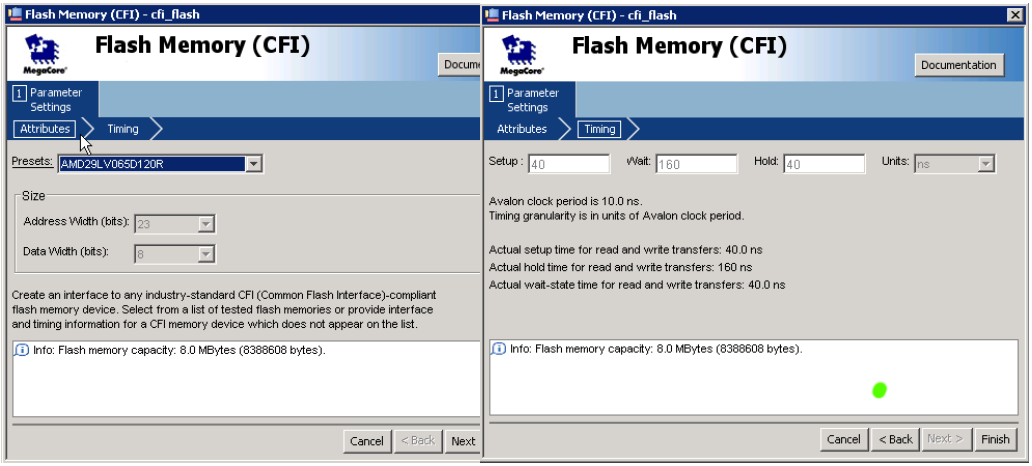
Adding the Avalon-MM Tristate Bridge

In the **Avalon-MM Tristate Bridge** configuration wizard, check the **Registered inputs and outputs** option to maximize system f_{MAX}, which increases the read latency by two for both the SRAM and flash memory.

Adding the Flash Memory Interface

The flash memory is 8M × 8-bit, which requires 23 address bits and 8 data bits. Figure 8–12 shows the **Flash Memory (Common Flash Interface)** configuration wizard settings for the example design.

Figure 8–12. Flash Memory Configuration Wizard



Adding the SRAM Interface

The SRAM device is 256K × 32-bit, which requires 18 address bits and 32 data bits. The example design uses a custom memory interface created with the SOPC Builder component editor. Figures 8–13 through 8–18 shows the settings required on the various component editor tabs to implement an interface to this SRAM.

Figure 8–13. SRAM Interface Component Editor HDL Files Tab

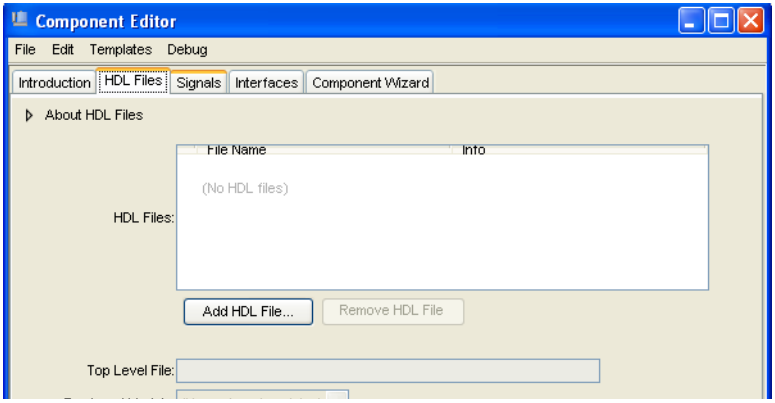


Figure 8–14. SRAM Interface Component Editor Signals Tab

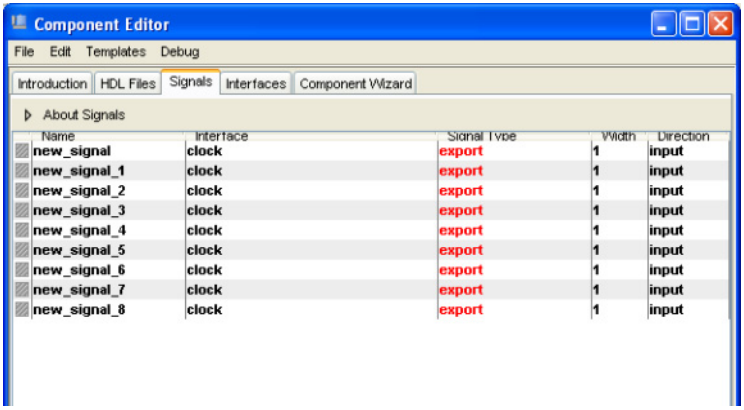


Figure 8–15. SRAM Interface Component Editor Interfaces Tab

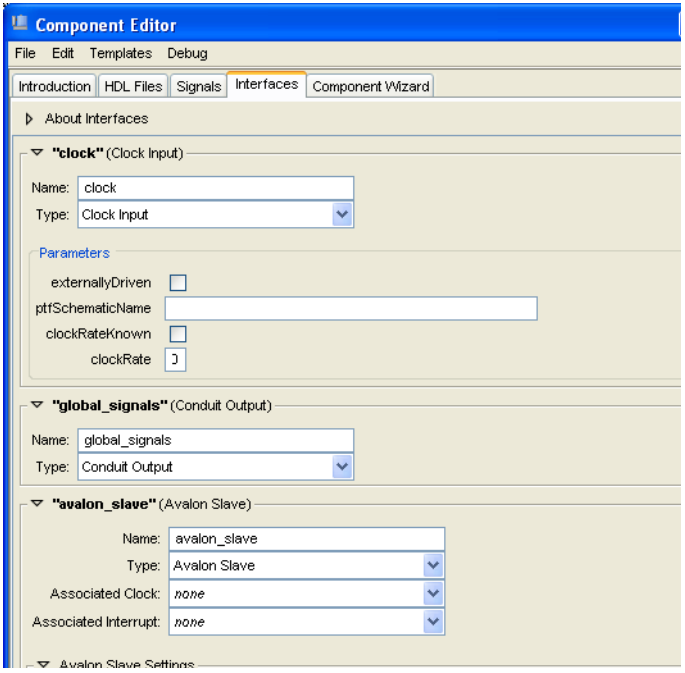
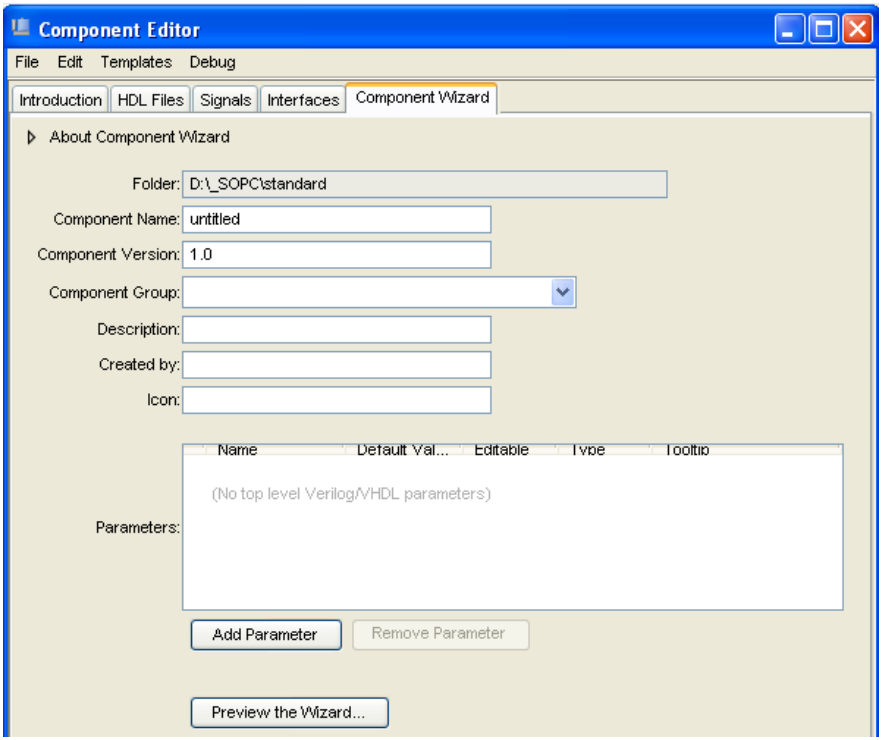


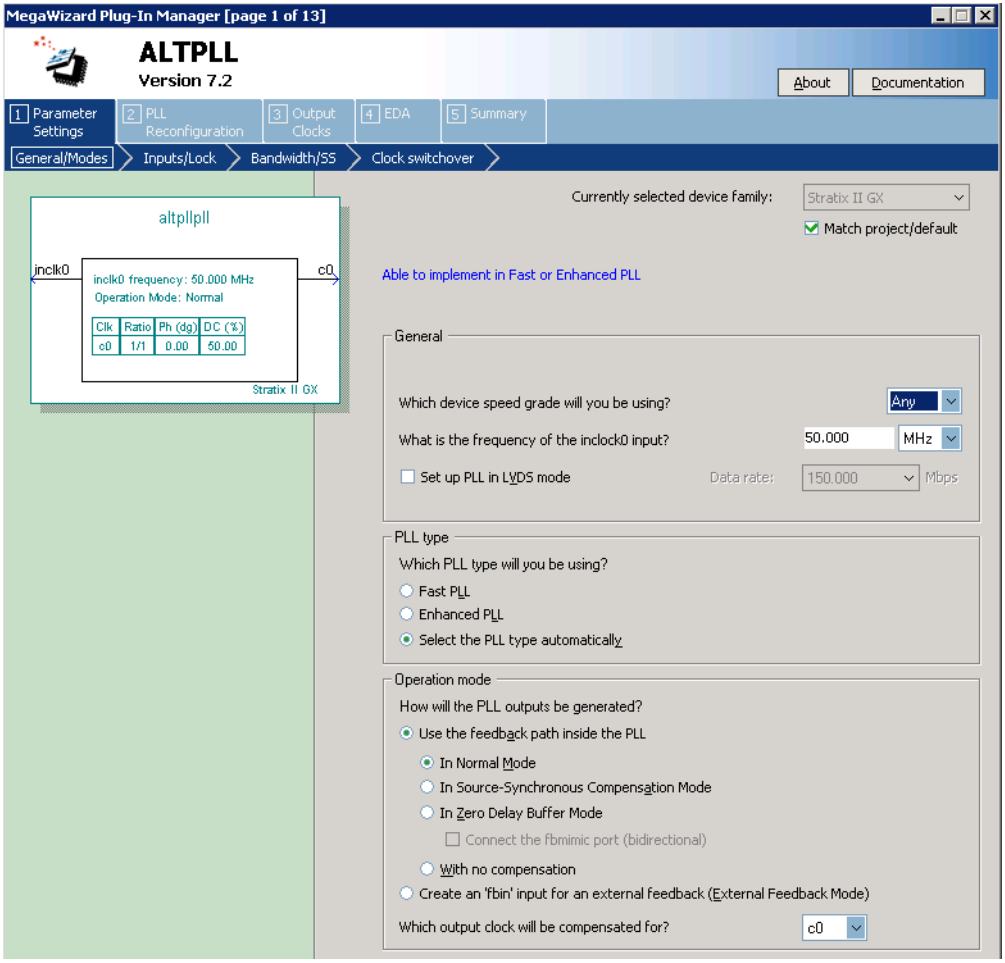
Figure 8–16. SRAM Interface Component Editor Component Wizard Tab



Adding the PLL

To reduce clock skew, all components in this example design connect to `sys_clk` generated by the **PLL** component. Select the **PLL** from the list of available components. To configure the **PLL**, select **Launch Altera's ALTPLL MegaWizard**. For this example design you configure `pll.c0` as a 50 MHz clock. Figure 8–17 illustrates the configuration of this component.

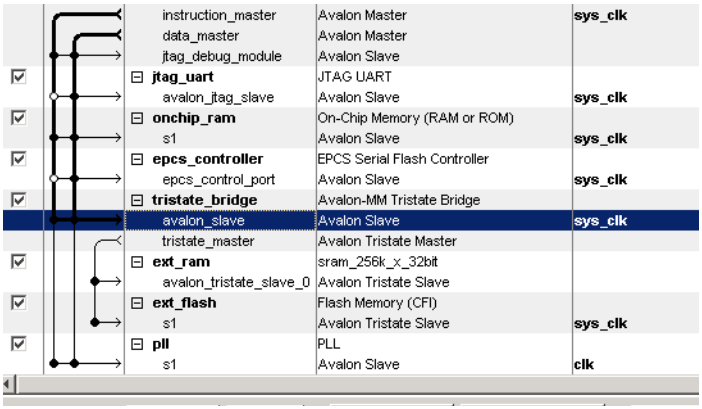
Figure 8–17. PLL Parameters



SOPC Builder System Contents Tab

Figure 8–18 shows the SOPC Builder system after adding the **Tristate bridge and memory interface** components, and configuring them appropriately on the **System Contents** tab. Figure 8–18 represents the complete example design in SOPC Builder.

Figure 8–18. SOPC Builder System with SRAM and Flash Memory



After generating the system, the top-level system module file **sopc_memory_system.v** contains the list of I/O signals for SRAM and flash memory that must be connected to FPGA pins:

```
output            chipselect_n_to_the_ext_ram;
output            read_n_to_the_ext_ram;
output            select_n_to_the_ext_flash;
output   [ 22: 0] tri_state_bridge_address;
output   [  3: 0] tri_state_bridge_byteenablen;
inout    [ 31: 0] tri_state_bridge_data;
output            tri_state_bridge_readn;
output            write_n_to_the_ext_flash;
output            write_n_to_the_ext_ram;
```

The Avalon-MM tristate bridge signals that can be shared are named after the instance of the tristate bridge component, such as `tri_state_bridge_data[31:0]`.

Connecting and Assigning Pins in the Quartus II Project

Figure 8–19 shows the result of generating the system module for the complete example design.

Figure 8–19. System Module with SDRAM and External Flash Memory



Figure 8–20 shows the pin assignments in the Quartus II assignment editor for some of the SRAM and flash memory pins. The correct pin assignments depend on the target board.

Figure 8–20. Pin Assignments for SRAM and Flash Memory

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reser
243	SRAM_BE_N[0]	PIN_M18	3	LVTTL	Column I/O		
244	SRAM_BE_N[1]	PIN_F17	3	LVTTL	Column I/O		
245	SRAM_BE_N[2]	PIN_J18	3	LVTTL	Column I/O	RUP3	
246	SRAM_BE_N[3]	PIN_L17	3	LVTTL	Column I/O	CLK15n	
247	SRAM_CS_N	PIN_B24	3	LVTTL	Column I/O	DQ9T4	
248	SRAM_OE_N	PIN_B26	3	LVTTL	Column I/O	DQ9T7	
249	SRAM_WE_N	PIN_C24	3	LVTTL	Column I/O	DQ59T	

Connecting FPGA Pins to Devices on the Board

Table 8–2 shows the mapping between the Avalon-MM address lines and the address pins on the SRAM and flash memory devices.

Table 8–2. FPGA Connections to SRAM and Flash Memory

Avalon-MM Address Line	Flash Address (8M × 8-bit Data)	SRAM Address (256K × 32-bit data)
tri_state_bridge_address[0]	A0	No connect
tri_state_bridge_address[1]	A1	No connect
tri_state_bridge_address[2]	A2	A0
tri_state_bridge_address[3]	A3	A1
tri_state_bridge_address[4]	A4	A2
tri_state_bridge_address[5]	A5	A3
tri_state_bridge_address[6]	A6	A4
tri_state_bridge_address[7]	A7	A5
tri_state_bridge_address[8]	A8	A6
tri_state_bridge_address[9]	A9	A7
tri_state_bridge_address[10]	A10	A8
tri_state_bridge_address[11]	A11	A9
tri_state_bridge_address[12]	A12	A10
tri_state_bridge_address[13]	A13	A11
tri_state_bridge_address[14]	A14	A12
tri_state_bridge_address[15]	A15	A13
tri_state_bridge_address[16]	A16	A
tri_state_bridge_address[17]	A17	A15
tri_state_bridge_address[18]	A18	A16
tri_state_bridge_address[19]	A19	A17
tri_state_bridge_address[20]	A20	No connect
tri_state_bridge_address[21]	A21	No connect
tri_state_bridge_address[22]	A22	No connect

Referenced Documents

This chapter references the following documents:

- *Introduction to Quartus II Manual*
- *Introduction to SOPC Builder*
- *SOPC Builder Components*
- *System Interconnect Fabric for Memory-Mapped Interfaces*
- *Avalon Memory-Mapped Interface Specification*
- Altera *Configuration Handbook*
- *Nios II Flash Programmer User Guide*
- *SDRAM Controller Core*
- *altpll Megafunction User Guide*
- *Common Flash Interface Controller Core*
- *Component Editor*

Document Revision History

Table 8–3 shows the revision history for this chapter.

Table 8–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007, v7.2.0	• Corrected Figure 8–19 to show flash memory changed example to use a PLL that is part of the SOPC Builder system, rather than a Quartus II component. Added section showing parameterization of PLL. (ADoQS Issue 1-5M4EN5 Lissy)	—
May 2007, v7.1.0	• Chapter 8 was previously chapter 9. • Updated Avalon terminology because of changes to Avalon technologies. Changed old "Avalon switch fabric" term to "system interconnect fabric." Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface." • Added section on Non-Default Memory Initialization. • On-chip Memory size, first parameter changed from Memory Width to Data Width and widths of 256, 512 and 1024 were added. • Corrected figure 8-18. • Added links to all referenced documents. • Removed discussions of reference designators for components because they are no longer required by SOPC Builder. • Removed unnecessary screenshots.	Updated to reflect changes to SOPC Builder for 7.1.0. SOPC Builder and improve readability.
March 2007, v7.0.0	No change from previous release.	—
November 2006, v6.1.0	No change from previous release.	—

Table 8–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
May 2006, v6.0.0	Chapter 9 was previously chapter 8. No change to content.	—
October 2005, v5.1.0	Chapter 8 was previously chapter 6. No change to content.	—
May 2005, v5.0.0	Initial release.	—

9. Developing Components for SOPC Builder

QII54007-7.2.1

Introduction

This chapter describes the design flow to develop a custom SOPC Builder component. The chapter describes the parts of a custom component and provides tutorial steps that guide you through the process of creating a custom component, integrating it into a system, and testing it in hardware.

This chapter is divided into the following sections:

- "Component Development Flow" on page 9–3.
- "Design Example: Checksum Master" on page 9–9. This design example demonstrates developing a component with both Avalon® Memory-Mapped (Avalon-MM) master and slave ports.
- "Sharing Components" on page 9–29. This section shows you how to use components in other systems, or share them with other designers.

SOPC Builder Components and the Component Editor

Typically, an SOPC Builder component is composed of the following four parts:

- HDL files that define the component's functionality as hardware.
- **_hw.tcl** file that describes the SOPC Builder related characteristics, such as interface behaviors.
- C-language files that define the component register map and driver software that allows programs to control the component if the component is accessed by a processor using software.

The component editor guides you through the creation of a module or **hw.tcl** file to describe your component. By following the procedures described in this document, you learn to use the component editor and turn any custom logic module into an SOPC Builder component.

After your component has been created, you can instantiate it in an SOPC Builder system and make connections in the same manner as other SOPC Builder components. You can share your component with other designers to encourage design reuse.

Prerequisites

This chapter assumes that you are familiar with the following:

- Building systems with SOPC Builder. For details, refer to the *Introduction to SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.
- SOPC Builder components. For details, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon-MM interface.

Hardware and Software Requirements

To use the design example in this chapter, you must have the following:

- Design files for the example design—A hyperlink to the design files appears next to the chapter, *Developing Components for SOPC Builder*, on the SOPC Builder literature page.
- Quartus® II Software version 7.2 or higher—Both Quartus II Web Edition and the fully licensed version will work with the example design.
- Nios® II Embedded Design Suite (EDS) version 1.1 or higher—Both the evaluation edition and the fully licensed version will work with the example design.
- Nios development board and an Altera® USB-Blaster™ download cable (Optional)—You can use any of the following Nios development boards:
  - Stratix® III Edition
  - Stratix® II Edition
  - Stratix Edition
  - Stratix Professional Edition
  - Cyclone® III Edition
  - Cyclone II Edition
  - Cyclone™ Edition

If you do not have a development board, you can follow the hardware development steps, but you cannot download the complete system to a working board.



You can download the Quartus II Web Edition software and the Nios II EDS, Evaluation Edition for free from the Altera Download Center at **www.altera.com**.

Component Development Flow

This section provides an overview of the development process for custom SOPC Builder components.

Typical Design Steps

A typical development sequence for an SOPC Builder component includes the following items:

1. Specification and definition.
    a. Define the functionality of the component.
    b. Determine the number and type of component interfaces, whether or not Avalon MM, Avalon ST, interrupt, or the interfaces that are used.
    c. Determine the component clocking requirements; what interfaces are synchronous to what clock inputs.
    d. If you want a microprocessor to control the component, specify the application program interface (API) to access and control the hardware.
    e. Specify the hardware functionality.
    f. If you want a microprocessor to control the component, specify the register set and application program interface (API) to access and control the component.
2. For hardware development, create an HDL file that describes the hardware in either Verilog or VHDL, and test the component alone in simulation or hardware to verify correct operation.
3. SOPC Builder import.
    a. Use the component editor to create an **hw.tcl** file that describes the component.
    b. Instantiate the component into a simple SOPC Builder system.
    c. Test register-level accesses to the component in hardware or simulation using a microprocessor, such as the Nios II processor.

When importing an HDL file into the component editor, any parameter definitions that are dependent upon other defined parameters cause an error. For example the following *DEPTH* parameter, though legal Verilog HDL syntax in the Quartus II software, causes an error in the component editor syntax checker:

```
parameter WIDTH = 32;
parameter DEPTH = ((WIDTH == 32) ? 8 : 16);
```

To avoid this error, use *localparam* for the dependent parameter instead, as shown below:

```
parameter WIDTH = 32;
localparam DEPTH = ((WIDTH == 32)?8:16);
```

4. Software Driver Development.
    a. Create a C header file that defines the hardware-level register map for software if the component is accessed by software.
    b. Write the driver software.
5. Finalize the component and distribute it for design reuse.

The following sections provide more details about the hardware and software design steps.

Hardware Design

As with any logic design process, the development of SOPC Builder component hardware begins after the specification phase. Creating the HDL design is an iterative process, as you write and verify the HDL logic against the specification.

The architecture of a typical component consists of the following functional blocks:

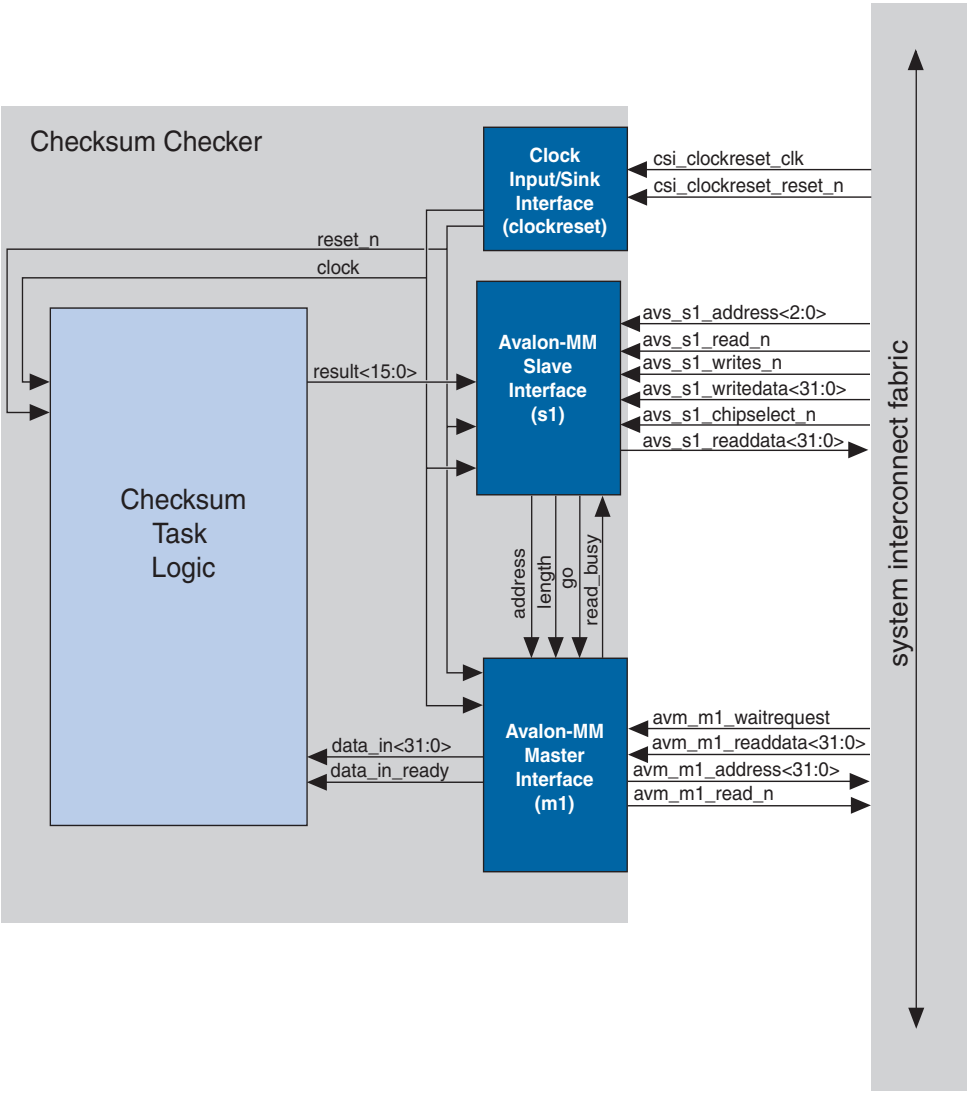
- *Task Logic*—Implements the component's fundamental function. The task logic is design dependent.
- *Interfaces*—Provide a standard way of providing data to or getting data from the components and of controlling the functioning of the components.

For interface specifications, refer to the following at **www.altera.com**:

- *Avalon Memory-Mapped Interface Specification*—Accommodate peripheral development for the SOPC environment.
- *Avalon Streaming Interface Specification*—Accommodate the development of high bandwidth low latency components for the SOPC environment.

Figure 9–1 shows the top-level blocks of a checksum component, which includes both Avalon-MM master and slave ports.

Figure 9–1. Checksum Component with Avalon-MM Master and Slave Ports



Software Design

If you want a microprocessor to control your component, then you must provide software files that define the software view of the component. At a minimum, you must define the register map for each Avalon MM slave port that is accessible to a processor.

Typically, the header file declares macros to read and write each register in the component, relative to a symbolic base address assigned to the component. The following example shows the register map of the checksum component for use by the Nios II processor.

Example 9–1. Example: Register Map for the Checksum Component

```
#ifndef __ALTERA_AVALON_CHECKSUM_REGS_H__
#define __ALTERA_AVALON_CHECKSUM_REGS_H__

#include <io.h>

/* Basic address, read and write macros. */

#define IOADDR_ALTERA_AVALON_CHECKSUM_ADDR(base)
__IO_CALC_ADDRESS_NATIVE(base, 0)
#define IORD_ALTERA_AVALON_CHECKSUM_ADDR(base)                IORD(base, 0)
#define IOWR_ALTERA_AVALON_CHECKSUM_ADDR(base, data)            IOWR(base, 0, data)

#define IOADDR_ALTERA_AVALON_CHECKSUM_LENGTH(base)
__IO_CALC_ADDRESS_NATIVE(base, 1)
#define IORD_ALTERA_AVALON_CHECKSUM_LENGTH(base)                IORD(base, 1)
#define IOWR_ALTERA_AVALON_CHECKSUM_LENGTH(base, data)          IOWR(base, 1, data)

#define IOADDR_ALTERA_AVALON_CHECKSUM_CTRL(base)
__IO_CALC_ADDRESS_NATIVE(base, 2)
#define IORD_ALTERA_AVALON_CHECKSUM_CTRL(base)                  IORD(base, 2)
#define IOWR_ALTERA_AVALON_CHECKSUM_CTRL(base, data)            IOWR(base, 2, data)

#define IOADDR_ALTERA_AVALON_CHECKSUM_RESULT(base)
__IO_CALC_ADDRESS_NATIVE(base, 4)
#define IORD_ALTERA_AVALON_CHECKSUM_RESULT(base)                IORD(base, 4)

#define IOADDR_ALTERA_AVALON_CHECKSUM_STATUS(base)
__IO_CALC_ADDRESS_NATIVE(base, 5)
#define IORD_ALTERA_AVALON_CHECKSUM_STATUS(base)                IORD(base, 5)


/* Masks. */

#define ALTERA_AVALON_CHECKSUM_CTRL_GO_MSK                      (0x1)
#define ALTERA_AVALON_CHECKSUM_STATUS_DONE_MSK                  (0x2)
#define ALTERA_AVALON_CHECKSUM_LENGTH_MSK                       (0xFFFF)
#define ALTERA_AVALON_CHECKSUM_RESULT_MSK                       (0xFFFF)

/* Offsets. */

#define ALTERA_AVALON_CHECKSUM_CTRL_GO_OFST                     (0)
#define ALTERA_AVALON_CHECKSUM_STATUS_BSY_OFST                  (0)
#define ALTERA_AVALON_CHECKSUM_STATUS_DONE_OFST                 (1)

#endif /* __ALTERA_AVALON_CHECKSUM_REGS_H__ */
```

Software drivers abstract hardware details of the component so that software can access the component at a high level. The driver functions provide the software an API to access the hardware. The software requirements vary according to the needs of the component. The most common types of routines initialize the hardware, read data, and write data.

When developing software drivers, it is instructive to look at the software files provided for other ready-made components. The Nios II EDS provides many components you can use as reference. See the <*Nios II EDS install path*>/**components**/ directory for examples.



For details on writing drivers for the Nios II hardware abstraction layer (HAL), refer to the *Nios II Software Developer's Handbook*.

Verifying the Component

You can verify the component in incremental stages, as you complete more of the design. Typically, you first verify the hardware logic as a unit (which might consist of multiple smaller stages of verification), and later you verify the component in a system.

Unit Verification

To test the task logic block alone, you use your preferred verification method(s), such as HDL simulation tools.

After you package the HDL files into a component using the component editor, the Nios II EDS offers an easy-to-use method to simulate read and write transactions to the component. Using the Nios II processor's robust simulation environment, you can write C code for the Nios II processor that initiates read and write transfers to your component. You can verify the results either on the ModelSim simulator or on hardware, such as a Nios development board.



For more information, refer to *AN 351: Simulating Nios II Embedded Processor Designs*.

System-Level Verification

After you package an **hw.tcl** file with the component editor, you can instantiate the component in a system, and verify the functionality of the overall system module.

SOPC Builder provides support for system-level verification for HDL simulators such as ModelSim. SOPC Builder automatically produces a test bench for system-level verification.



You can include a Nios II processor in your system to enhance simulation capabilities during the verification phase. Even if your component has no relationship to the Nios II processor, the auto-generated ModelSim simulation environment provides an easy-to-use starting point.

Design Example: Checksum Master

This section uses a **checksum master** design example to demonstrate the steps to create a component and instantiate it in a system. This component includes both Avalon-MM master and slave ports.

In this section, you will perform the following steps:

1. Install the design files.
2. Review the example design specifications.
3. Create an SOPC Builder component.
4. Instantiate the component in an SOPC system.
5. Compile the hardware design in the Quartus II software, and download the design to a target board.
6. Exercise the hardware using the Nios II processor.

Install the Design Files

Before you proceed, you must install the Nios II development tools and download the **checksum master** example design from the Altera website. The hardware design used in this chapter is based on the **standard** hardware example design included with the Nios II EDS.

Perform the following steps to set up the design environment:

1. On your host computer file system, locate the following directory:

   *<Nios II EDS install path>***/examples/***<verilog or vhdl>***/***<board version>***/standard**

Each development board has a VHDL and Verilog HDL version of the design. You can use either of these design examples. Table 9–1 shows the names of the directories for each Nios development board.

Table 9–1. Design File Directories

Nios Development Board	Design Directory
Stratix III Edition	niosII_stratixIII_3sl150
Stratix II Edition	niosII_stratixII_2s60_ROHS, niosII_stratixII_2s60, niosII_stratixII_2s60ES
Stratix Edition	niosII_stratix_1s10, niosII_stratix_1s40
Stratix Professional Edition	niosII_stratix_1s40
Cyclone III Edition	niosII_cycloneIII_3c120, niosII_cycloneIII_3c25
Cyclone II Edition	niosII_cycloneII_2c35
Cyclone Edition	niosII_cyclone_1c20

2. Copy the **standard** directory to a new location. By copying the design files, you avoid corrupting the original design and avoid issues with file permissions. This document refers to the newly-created directory as the **<*Quartus II project*>** directory.

3. Copy the file **altera_avalon_checksum.zip** to the <*Quartus II project*> directory and unzip it. The design and test files listed in Table 9–2 are added to <*Quartus II project*>**/altera_avalon_checksum** directory.

Review the Example Design Specifications

This section discusses the design specifications for the provided checksum example design, giving details on each of the following topics:

- Checksum Design Files
- Functional Specification
- Master Task Logic
- Register File
- Avalon-MM Master Interface
- Avalon-MM Slave Interface
- Software API

Checksum Design Files

Table 9–2 lists the contents provided in the **altera_avalon_checksum** directory.

Table 9–2. Checksum Design Files Directory

File Name	Description
/altera_avalon_checksum	Contains all the HDL and software files for the component. All the HDL files must be in the same directory and be consistent in name with the **hw.tcl** file.*(1)*
altera_avalon_checksum.v	The top-level HDL file instantiates the task logic, Avalon-MM master and slave interfaces and the register files.
checksum_task_logic.v	This Verilog HDL file contains the core functionality of the checksum component.
read_master.v	This file contains the logic for the Avalon-MM read master interface.
s1_slave.v	This file contains logic for reading and writing to the checksum registers
altera_avalon_checksum_sw.tcl	This is the checksum software driver configuration file for the Nios II command line flow.
/inc	This sub-directory includes header files defining the low-level hardware interface.
altera_avalon_checksum_regs.h	This file defines macros to access registers in the checksum component.
/test_software	This sub-directory includes an example program to test the component hardware and software.
test_checksum.c	The test program initializes and array of data for the checksum component to read and compute the checksum.

Note to Table 9–2:
(1) The component editor creates the **altera_avalon_checksum_hw.tcl** file and stores it in the **altera_avalon_checksum** directory.

Master Task Logic

The **checksum master** reads a programmable number of 16-bit values to calculate a checksum. The `status` register sets its `DONE` bit when the checksum master completes. Software polls the `DONE` bit to determine when the calculation is complete.

Register File

The register file provides access to the `configuration`, `status`, and `results` registers shown in Table 9–3. The design maps each register to a unique offset in the Avalon-MM slave port address space. The registers are read, write, or read only.

Table 9–3. Register File and Address Mapping of Checksum Master

Register Name	Offset	Access	Description
`Address`	0x00	Read/Write	32-bit start address for checksum calculations.
`Length`	0x04 + 4	Read/Write	16-bit byte count for the checksum calculation.
`Control`	0x08 + 8	Read/Write	Bits [7:1] are reserved. Bit[0] is the GO bit.
`Reserved`	0x0C + 12	—-	—
`Result`	0x10 + 16	Read	16-bit result of the checksum calculation.
`Status`	0x14 + 20	Read	Bits [7:2] are reserved. Bit[1:0] are DONE and BUSY.
`Reserved`	0x18	—	—
`Reserved`	0x1C	—	—

Table 9–4 shows the layout of the bits and fields of these registers.

Table 9–4. Layout of Checksum Master Registers

Offset	31 16	15 1	0
0x00	address		
0x04	reserved	length	
0x08	reserved		GO
0x10	reserved	result	
0x14	reserved	DONE	BUSY

Avalon-MM Clock Interface

The **checksum** component includes an Avalon-MM clock interface to bring in a system clock and reset into the checksum component as shown in Figure 9–1. The clock interface will be connected to each Avalon-MM master and slave interface in the **Interface** tab.

Table 9–5 lists the clock interface signals that comprise the Avalon-MM master port.

Table 9–5. Table of Clock Interface Signals

Signal Name in HDL	Avalon-MM Signal Type	Width	Dir	Notes
csi_clockreset_clk	clk	1	In	Synchronization clock for the component. All signals are synchronous to clk.
csi_clockreset_reset_n	reset_n	1	In	Resets the entire Avalon-MM system.

Avalon-MM Master Interface

The **checksum master** component includes an Avalon-MM master port that reads from memory. The component's Avalon-MM master port has the following characteristics:

- It is synchronous to the Avalon-MM master clock interface.
- It initiates master transfers to the system interconnect fabric.

Table 9–6 lists the signals that comprise the Avalon-MM clock port.

Table 9–6. Table of Checksum Avalon-MM Master Port Signal Names and Avalon Signal Types

Signal Name in HDL	Avalon-MM Signal Type	Width	Dir	Notes
avm_m1_address	address	32	Out	Byte address aligned on word boundary.
avm_m1_byteenable	byteenable	4	Out	Enables specific byte lanes on ports greater than 8 bits.
avm_m1_read_n	read_n	1	Out	Read request signal.
avm_m1_readdata	readdata	32	In	Uni-directional data.
avm_m1_waitrequest	waitrequest	1	In	Forces master port to wait until the system interconnect fabric is ready to proceed with the transfer.

Avalon-MM Slave Interface

The Avalon-MM slave port handles simple read and write transfers to the registers. The slave port has the following characteristics:

- Synchronous to the Avalon-MM clock interface.

- Readable and writable.
- Zero wait states for writing and one wait state for reading.
- No setup or hold restrictions for reading and writing.
- Uses native address alignment, because the slave port is connected to registers rather than a memory device.

Table 9–7. Table of Checksum Avalon-MM Slave Port Signal Names and Avalon Signal Types

Signal Name in HDL	Avalon-MM Signal Type	Width	Dir	Notes
avs_s1_address	address	3	In	A byte address.
avs_s1_read_n	read_n	1	In	Read request input.
avs_s1_write_n	write_n	1	In	Write request input.
avs_s1_chipselect_n	chipselect	1	In	Chip-select to slave port. Slave port ignores all other signals unless it is selected.
avs_s1_readdata	readdata	32	Out	Uni-directional read data
avs_s1_writedata	writedata	32	In	Uni-directional write data

Software API

The **altera_avalon_checksum_regs.h** file has been provided to include macros to read and write the checksum slave registers.

Create an SOPC Builder component

In this section you specify the hardware interfaces to the component, and define the behavior of each interface signal.

Open the Quartus II Project and Start the Component Editor

To open SOPC Builder from the Quartus II software, perform the following steps:

1. Start the Quartus II software.
2. Open the project **standard.qpf** in the <*Quartus II project*> directory.
3. On the Tools menu, click **SOPC Builder**. SOPC Builder appears, displaying a ready-made example design containing a Nios II processor and several components.
4. On the File menu, click **New Component**. The component editor appears, displaying the **Introduction** tab.

HDL Files Tab

In this section you associate the component's top-level HDL file with the component's hardware Tcl file using the **HDL files** tab. Perform the following steps:

1. Click the **HDL Files** tab.
2. Click **Add HDL File**.
3. Browse to the <*Quartus II project*>**/altera_avalon_checksum** directory and select the top level HDL file **altera_avalon_checksum.v** and click **Open**.



The first file you add to the component editor must be the top-level HDL file of your design.

4. Click **OK** when a message indicated analysis is complete.
5. You can now add lower-level design files. Click **Add HDL File** and add the **checksum_task_logic.v**, **read_master.v**, and **sl_slave.v** files to the component list.
6. Select the top level module of your component by clicking in the **Top Level Module** list and selecting **altera_avalon_checksum**.
7. If you plan to simulate your component, click **Add Simulation File** to add all of the files required for simulation.

The component editor now displays error messages. You are instructed to fix them in later steps.

Signals Tab

For every I/O signal present on the top-level HDL module, you must map the signal name to a valid signal type using the **Signals** tab. If the signal name includes a recognized signal type (such as `write` or `address`), the component editor guesses the signal's type. If the component editor cannot determine the signal type, it assigns the type `export`.

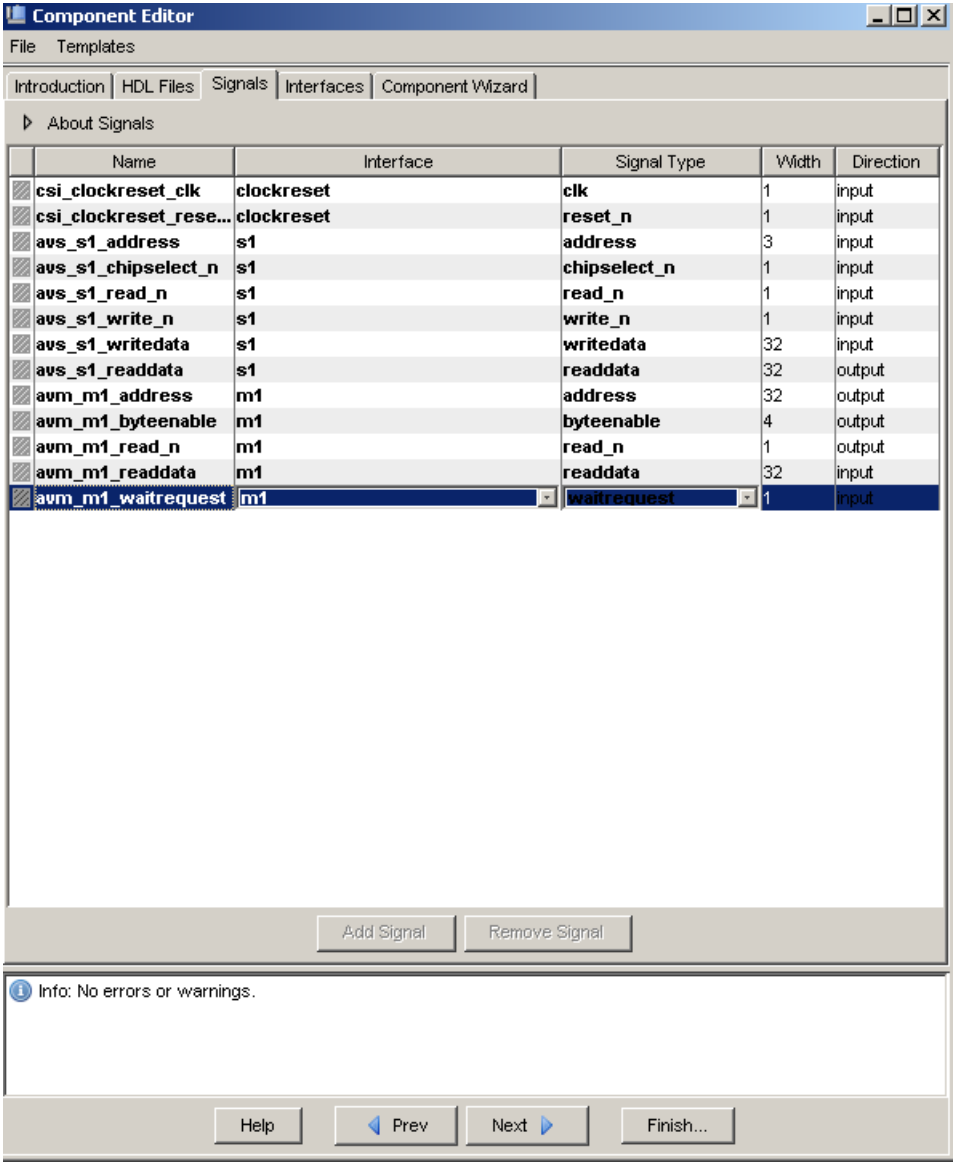
This design uses the automatic type and interface recognition feature of the component editor to quickly allow the component editor to assign the component signals to the appropriate interface and signal type. To change the type assigned, click at the right edge of the **Signal Type** column for the signal in question. A pull-down menu provides other choices.



For more information on the automatic type and interface recognition feature see the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*.

This design includes three interfaces: clock (clockreset), slave (s1), and master (m1) as illustrated in Figure 9–2. The signal types and polarities are derived from the signal names.

Figure 9–2. The Signals Tab



Interfaces Tab

After assigning signals to interfaces, the **Interfaces** tab allows you to further configure the properties of all interfaces on the component.

Perform the following steps to configure the Avalon slave port:

1. Click the **Interfaces** tab. The component editor displays the Avalon-MM slave port (`s1`) from the previous tab.

2. Remove any unused interfaces by clicking **Remove Interfaces with No Signals**.



This removes the default provided clock and export_0 interfaces in the component editor, as you created your own interfaces with the automatic type and interface recognition feature.

The component editor now displays the clockreset clock input interface, s1 slave interface, and the m1 master interface.

3. For the Avalon-MM slave port (s1) set the clock and reset for the slave interface by clicking on **Associated Clock** and then select **clockreset**.

4. Change the default settings for the slave port to match those given in Table 9–8.

Table 9–8. Settings for Avalon-MM Slave Port (Part 1 of 2)

Slave Settings	Value	Description
Slave Addressing	Native	Indicates that the slave ports uses address-mapped registers.
Minimum Arbitration Shares	1	Arbitration shares modify the default round-robin arbitration scheme which provides equal access to all devices.
Can receive stderr/stdout	No	—
Interleave Bursts	No	—
Read Latency	0	—
Max. Pending Read Transactions	0	—
Slave Timing	**Value**	**Description**
Setup	0	Indicates that the slave port responds to a read or write request in a single clock cycle.
Read Wait	1	Indicates that the slave port responds to read requests one cycle after they are made (one read waitstate).

Table 9–8. Settings for Avalon-MM Slave Port (Part 2 of 2)

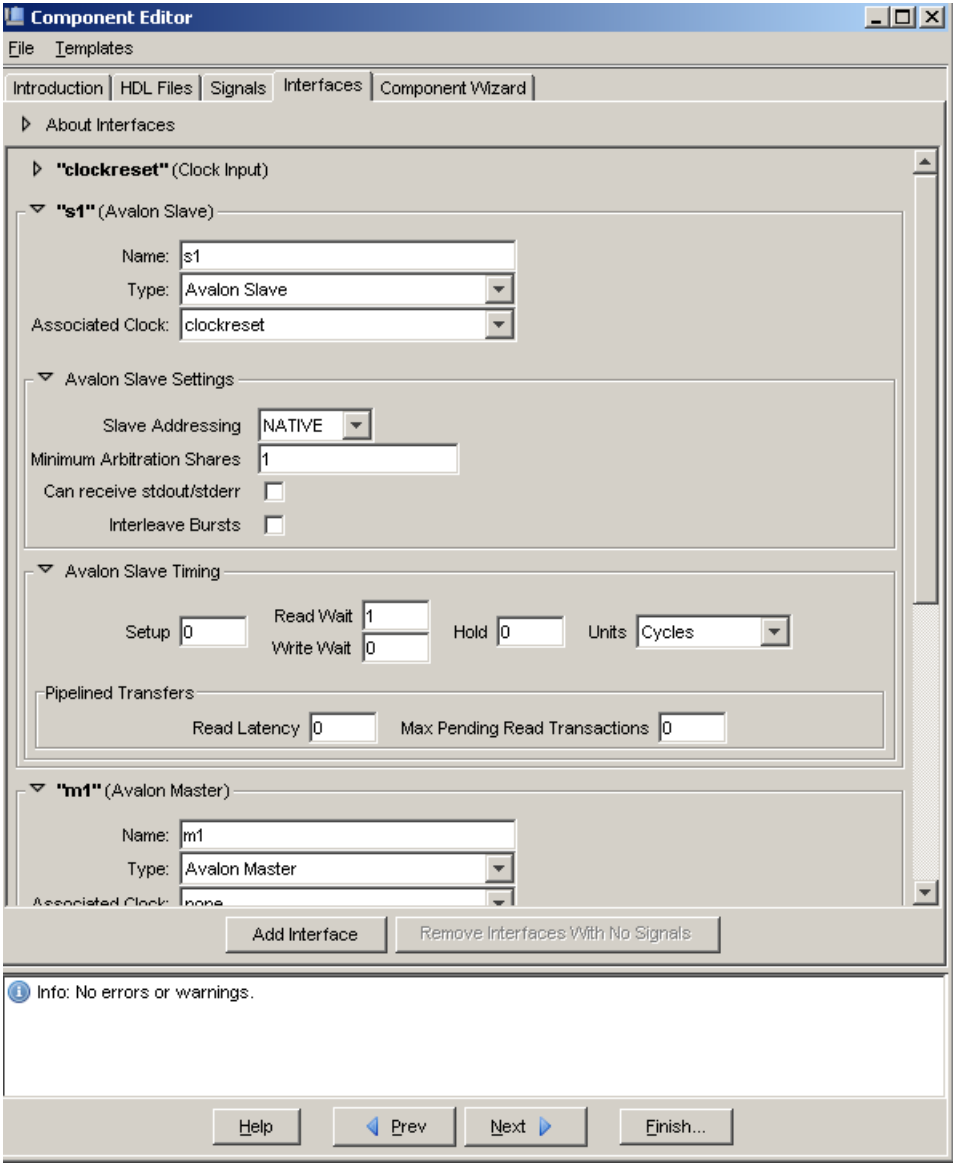
Slave Settings	Value	Description
Write Wait	0	Indicates that the slave port responds to write requests in a single clock cycle and does not need write waitstates.
Hold	0	Indicates that there is not a hold time requirement.

5. For the Avalon-MM master port (m1) set the clock and reset for the master interface by clicking on **Associated Clock** and then select **clockreset**.

6. Leave all other Avalon-MM master settings as the default settings, as shown in Figure 9–4.

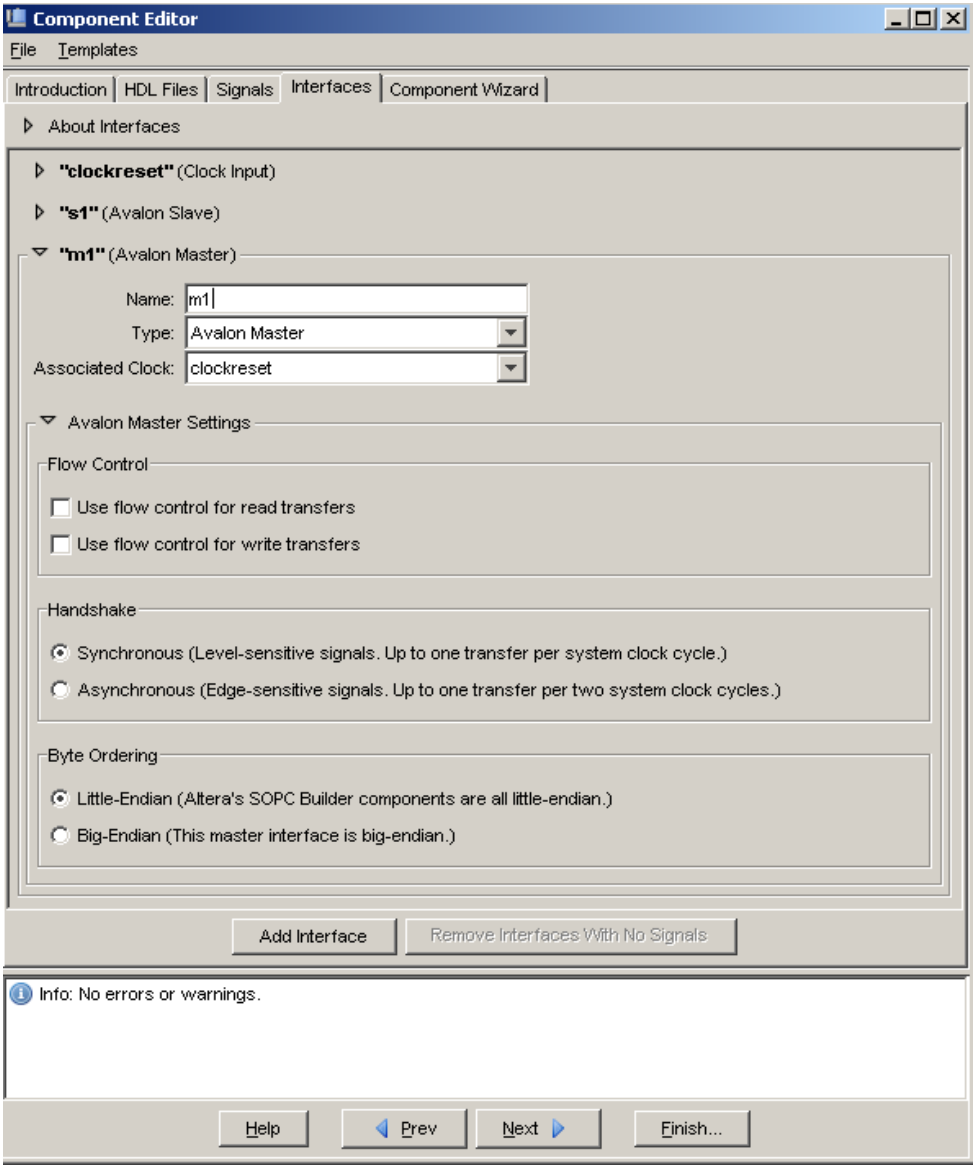
Figure 9–3 illustrates the slave settings.

Figure 9–3. Avalon-MM Slave Interfaces Settings



The Avalon-MM master port uses the default settings. Figure 9–4 illustrates these settings.

Figure 9–4. Avalon-MM Masters Interfaces Settings

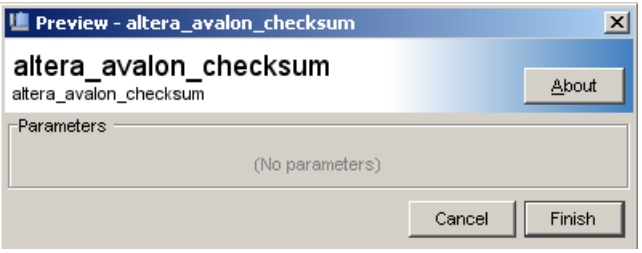


Component Wizard Tab

The **Component Wizard** tab allows you to control how SOPC Builder presents the components to a user. Perform the following steps to configure the user presentation of the component. The component editor creates a default name for the component, based on the name of the top-level design module.

1. Click the **Component Wizard** tab.
2. For this example, do not change the default settings for **Component Name** or **Component Version**.
3. For the **Component Group** type the following: `User Logic`
4. Complete the remaining fields, such as **Description** and **Created By**.
5. Click **Preview the Wizard** to preview the component wizard as it will appear in SOPC Builder. Figure 9–5 illustrates the component wizard preview.
6. Close the Preview window.

Figure 9–5. Component Wizard



Save the Component

Perform the following steps to save the component and exit the component editor:

1. Click **Finish**. A message describes the file that is created for the component.
2. Click **Yes** to save the file. The component editor saves the **altera_avalon_checksum_hw.tcl** file in the same directory that you stored the top-level component HDL file. The component editor closes, and you return to SOPC Builder.

3. Locate the new **checksum** component in the list of available components under the **User Logic** group. The component is added to the SOPC Builder search path. Right-clicking on a component in the list allows you to edit the component.

Instantiate the Component in Hardware

At this point, the new component is ready to instantiate in an SOPC Builder system. The remaining steps for this design example illustrate one possible method of instantiation that includes the following general steps:

1. Add the **checksum master** to the SOPC Builder system.
2. Compile the hardware design and download to the target board.

Add the checksum Master Component to the SOPC Builder System

Perform the following steps to add a **checksum master** component to the SOPC Builder system:

1. On the SOPC Builder **System Contents** tab, select the new component **altera_avalon_checksum** under the **User Logic** group in the list of available components, and click **Add**. The configuration wizard for the **checksum master** component appears.
2. Click **OK**. The component **altera_avalon_checksum_inst** appears in the table of active components.
3. Connect the **altera_avalon_checksum_inst** `m1` master port to a memory in your system.



The test program uses an on-chip memory peripheral called **onchip_ram**. If your SOPC Builder system does not have an on-chip memory you should add an on-chip memory to the design. The test program requires that the name of the on-chip RAM and the component name used in the test program match. Connect the on-chip RAM to the Nios II data master.

4. To start generating the system, click **Generate**
5. After system generation completes successfully, exit SOPC Builder.

Compile the Hardware Design and Download to the Target Board

At this point, you have created an SOPC Builder system that uses the **checksum** component. The **checksum** component adds no additional I/O signals to the SOPC Builder system top-level so you only need to compile the design in the Quartus II software.

Perform the following steps to compile the hardware design and download it to the target board.

1. On the Processing menu, click **Start Compilation** to start compiling the hardware design. The compilation begins.

   If you performed all prior steps correctly, the Quartus II compilation finishes successfully after several minutes, and generates a new SRAM Object File (**.sof**) for the project.



You can only perform the remaining steps in this chapter if you have a development board.

2. Connect your host computer to the development board using an Altera download cable, such as the USB Blaster, and apply power to the board.

3. On the Tools menu, click **Programmer** to open the Quartus II Programmer.

4. Use the Programmer window to download the following FPGA configuration file to the board: <*Quartus II project*>**/standard.sof**.

At this point, you have completed all the steps to create a hardware design and download it to hardware.

Exercise the Hardware Using Nios II Software

The **checksum master** example design is based on the Nios II processor. The example design files provide a C test program that programs the component to calculate a checksum and then polls the component to determine if it completes the calculation successfully. In this section you perform the following steps:

1. Start the Nios II IDE and create a new Nios II IDE project.

2. Build and run the C test program.

3. View the results.

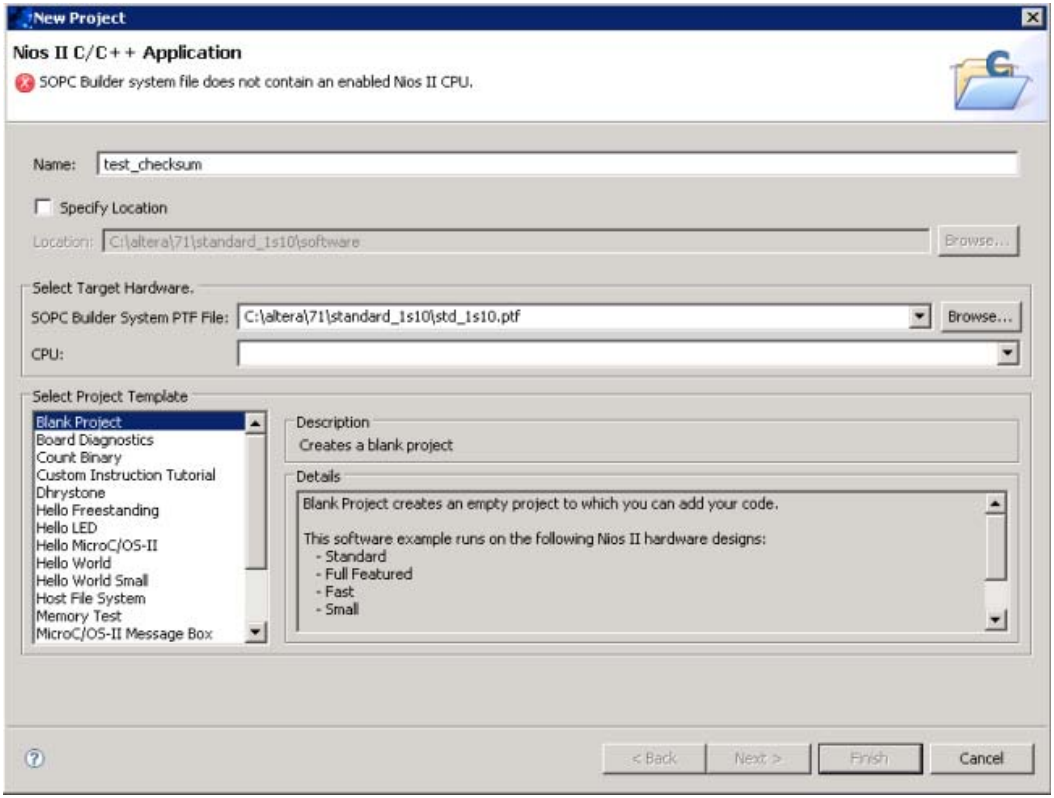
To complete this section, you must have performed all prior steps, and successfully configured the target board with the hardware design.

Start the Nios II IDE and Create a New IDE Project

Perform the following steps to start the Nios II IDE and create a new IDE project:

1. Start the Nios II IDE.
2. On the Window menu, point to **Open Perspective** and click **Other,** then click **Nios II C/C++** to open the Nios II C/C++ perspective.
3. On the File menu, point to **New** and then click **C/C++ Application** to start a new project. The first page of the **New Project** wizard appears.
4. Under **Select Project Template**, select **Blank Project**.
5. In the **Name** box, type `test_checksum`.
6. Ensure that **Specify Location** is turned off so that you use the default software directory under your standard board as shown in Figure 9–6.

Figure 9–6. Create a New Project



7. Click **Browse** under **Select Target Hardware**. The **Select Target Hardware** dialog box appears.
8. Browse to the <*Quartus II project*> directory.
9. Select the file **std**_<*FPGA*>**.ptf**.
10. Click **Open** to return to the New Project wizard. The **SOPC Builder System** and the **CPU** fields are now specified.
11. Click **Finish**. After the IDE successfully creates the new project, the C/C++ Projects view contains two new projects, **test_checksum** and **test_checksum_syslib**.

Compile the Software Project and Run on the Target Board

In this section you compile the C test program provided with the checksum design files, and then download it to the target board.

First, perform the following steps to associate the source files with the new C/C++ project:

1. Copy **test_checksum.c** from <*Quartus II project*>**/altera_avalon_checksum/test_software** to the <*Quartus II project*>**/software/test_checksum** directory.

2. In the Nios II IDE C/C++ Projects view, right-click **test_checksum** and click **Refresh**, directing the IDE to recognize the new file in the project directory.

The project is now ready to compile and run. Perform the following steps:

1. Right-click the project **test_checksum** in the Nios II C/C++ Projects view and click **Build Project** to compile the program. The first time you build the project, it can take a few minutes for the compilation to finish.

2. After compilation completes, select **test_checksum** in the C/C++ Projects view.

3. On the Run menu, click **Run**. The **Run** dialog box appears.

4. Select **Nios II Hardware**, and click **New**. A new run/debug configuration named **test_checksum Nios II HW configuration** appears.

5. If the **Run** button (in the bottom right of the **Run** dialog box) is disabled, perform the following steps:

    a. Click the **Target Connection** tab.

    b. Click **Refresh** next to the **JTAG cable** list.

    c. In the **JTAG cable** list, select the download cable you want to use.

    d. Click **Refresh** next to the **JTAG device** list.

6. Click **Run**.

7. View the results: The **Console** view in the IDE displays messages similar to the following: `0x5a5a`.

You have finished all steps for the **checksum** design example.

Sharing Components

When you create a component, component editor by default saves the (**_hw.tcl**) in the same directory as the top-level HDL file. Where appropriate, files referenced by the **_hw.tcl** file all use relative paths so that files can easily be moved and copied together. To promote design reuse, you can use the component in different projects, and you can share your component with other designers.

Perform the following steps to share a component:

1. In your computer's file system, move the component directory to a central location, outside any particular Quartus II project's directory. For example, you could create a directory **c:\my_component_library** to store your custom components.



If you create a new component library under the Quartus II project directory and then add individual components to that new component library, for example: <*Quartus_rootdir*>**\sopc_builder\my_project\my_project_lib\component1**, SOPC Builder cannot find the components. You must add the directory for **component1** to your library path.



SOPC Builder will find your components if you place your components in the **projectdir\ip** directory. Altera recommends that you do so.

2. On the Quartus II Assignments menu, click **Settings.** The **Settings** dialog box appears.

3. In the **Categories** list, click **Libraries**.

4. Under **Global libraries**, add the path to the enclosing directory of the component directory. For example, for a component directory **c:\my_component_library\checksum_master**, add the path **c:\my_component_library**.



If you need to share a component library directory across projects, you can ad items to the **SOPC Builder Tools\Options\IP Search Path** settings. However, in the 7.2 version of the Quartus II software, this specifies component directories, and not library directories.

To use the newly created component in another SOPC Builder system, you must perform one of the following:

- Copy the component and its related files into the IP subdirectory of the project where it is to be used. For example, to use the component in the **project 2** project, simply copy the Tcl File (**.tcl**) and the reference files to **project2/ip/checksum**, and they will be found automatically.
- Alternatively, you can place the Tcl File (**.tcl**) and related files elsewhere in a component library, such as **L:/components/checksum/**, and add the library location to see the search path via **SOPC Builder/Tools/Options/IP Search Path**.

Referenced Documents

This chapter references the following documents:

- *Introduction to SOPC Builder*
- *SOPC Builder Components*
- *The Component Editor*
- *Avalon Memory Mapped Interface Specification*
- *Nios II Software Developer's Handbook*
- *AN 351: Simulating Nios II Embedded Processor Designs*

Document Revision History

Table 9–9 shows the revision history for this chapter.

Table 9–9. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007, v7.2.1	Updated instructions on how to develop components to match updated GUI.	—
October 2007, v7.2.0	Updated instructions on how to develop components to match new GUI.	—
May 2007, v7.1.0	Changed example component from a pulse width modulator with that only has an Avalon-MM slave interface to a checksum master that includes both Avalon-MM master and slave interfaces.	Changed the example design to one with more practical applications. Updated instructions for the 7.1 release.
March 2007, v7.0.0	No change from previous release.	—
November 2006, v6.1.0	Chapter 9 was previously chapter 10. No change to content.	—
May 2006, v6.0.0	Chapter 10 was previously chapter 9. No change to content.	—
October 2005, v5.1.0	Chapter 9 was previously chapter 7. No change to content.	—
August 2005, v5.0.1	Corrected Table 7-5.	—
May 2005, v5.0.0	No change from previous release.	—
February 2005, v1.0	Initial release.	—

Section III. Interconnect Components

This section provides information on Avalon Memory-Mapped (Avalon-MM) and Avalon Streaming (Avalon-ST) components that can be added to SOPC Builder systems. The components described in these chapters help you to create and optimize your SOPC Builder system. They are provided for free and can be used without a license in any design targeting an Altera device.

This section includes the following chapters:

- Chapter 10, Avalon Memory-Mapped Bridges
- Chapter 11, Avalon Streaming Interconnect Components



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

10. Avalon Memory-Mapped Bridges

QII54020-7.2.0

Introduction to Bridges

This chapter introduces the concept of Avalon® Memory-Mapped (Avalon-MM) bridges, and describes the Avalon-MM bridge components provided by Altera® for use in SOPC Builder systems.

A bridge, in the context of SOPC Builder, is a component that acts as part of the system interconnect fabric. Bridges are not end-points for data, but rather affect the way data is transported between other components. By manually inserting Avalon-MM bridges between Avalon-MM master and slave ports in a system, you can control system topology, which in turn affects the interconnect that SOPC Builder generates. Manual control of the interconnect can result in higher performance and/or lower logic utilization.

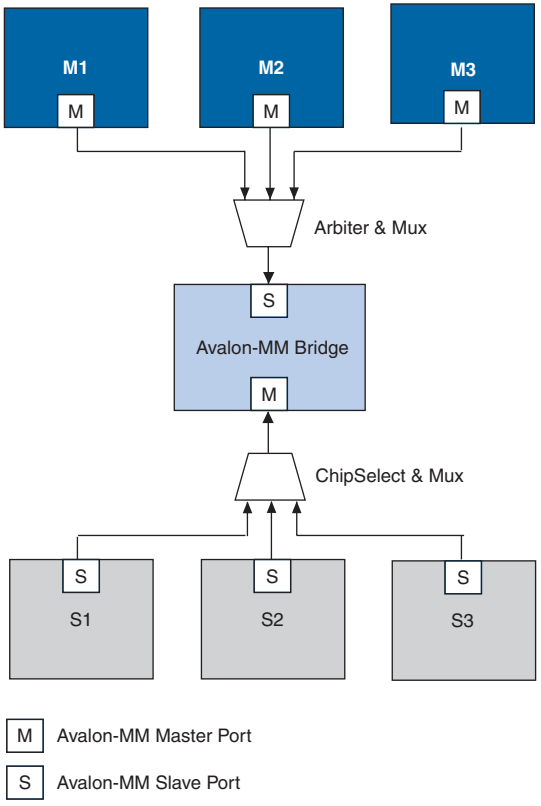
Altera provides the Avalon-MM bridge, which is described in this chapter:

- "Avalon-MM Pipeline Bridge" on page 10–9

Structure of a Bridge

A bridge has one Avalon-MM slave port and one Avalon-MM master port, as shown in Figure 10–1. In an SOPC Builder system, one or more master ports connect to the bridge's slave port to control the bridge. The bridge's master port connects, in turn, to one or more slave ports. You configure the master-slave pairs manually with the SOPC Builder GUI. In Figure 10–1, all three masters have a logical connection to all three slaves, although physically each master only connects to the bridge.

Figure 10–1. Example of an Avalon-MM Bridge in an SOPC Builder System



A bridge issues transfers on its master port in the same order in which they were received. Transfers initiated to the bridge's slave port propagate to the master port in the same order in which they were initiated on the slave port.

If you use either the Avalon-MM pipeline bridge or the Avalon-MM clock-crossing bridge in your system discussed in the SOPC Builder chapter, automatic pipelining feature is disabled.

For details on the Avalon-MM interface, refer to the *Avalon Memory-Mapped Interface Specification*.

Reasons for Using a Bridge

Reasons you might use an Avalon-MM bridge include:

- Increase the f_{MAX} of your system
- Control system topology
- Specify separate clock domains for master-slave pairs

If there are no bridges between master-slave pairs, SOPC Builder generates system interconnect fabric with maximum parallelism so that all masters can drive transactions to and from all slaves concurrently as long as each master is trying to access a different slave. This default behavior incurs the cost of additional arbiters and multiplexers decreasing the f_{MAX} of the system. For high performance systems that do not require a large degree of concurrency, the default behavior might not provide optimal performance. With knowledge of the system and application, you can optimize the system interconnect fabric by inserting bridges to control the system topology.

Figure 10–2 and Figure 10–3 show an SOPC system without bridges. This system includes three CPUs, a DDR SDRAM controller, a message buffer RAM, a message buffer mutex, and a tristate bridge to an external SRAM.

Figure 10–2. Example System Without Bridges — SOPC Builder View

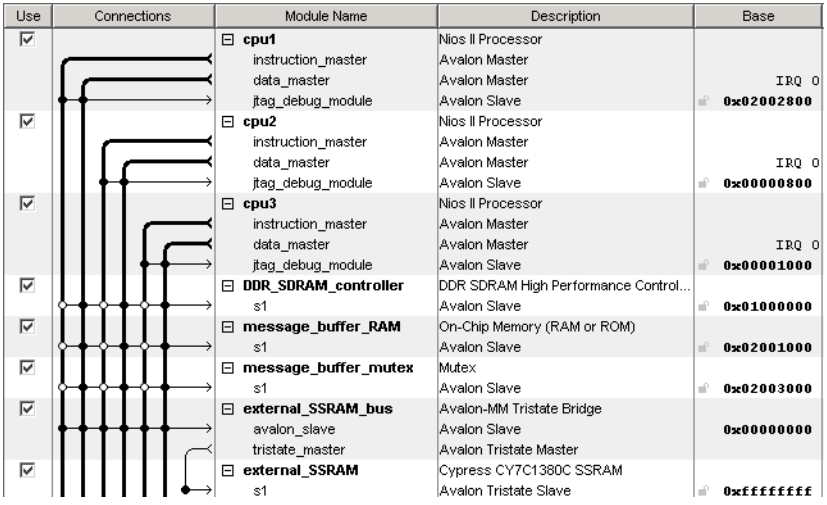


Figure 10–3 illustrates the default system interconnect fabric that SOPC Builder would create for the system in Figure 10–2.

Figure 10–3. Example System without Bridges - System Interconnect View

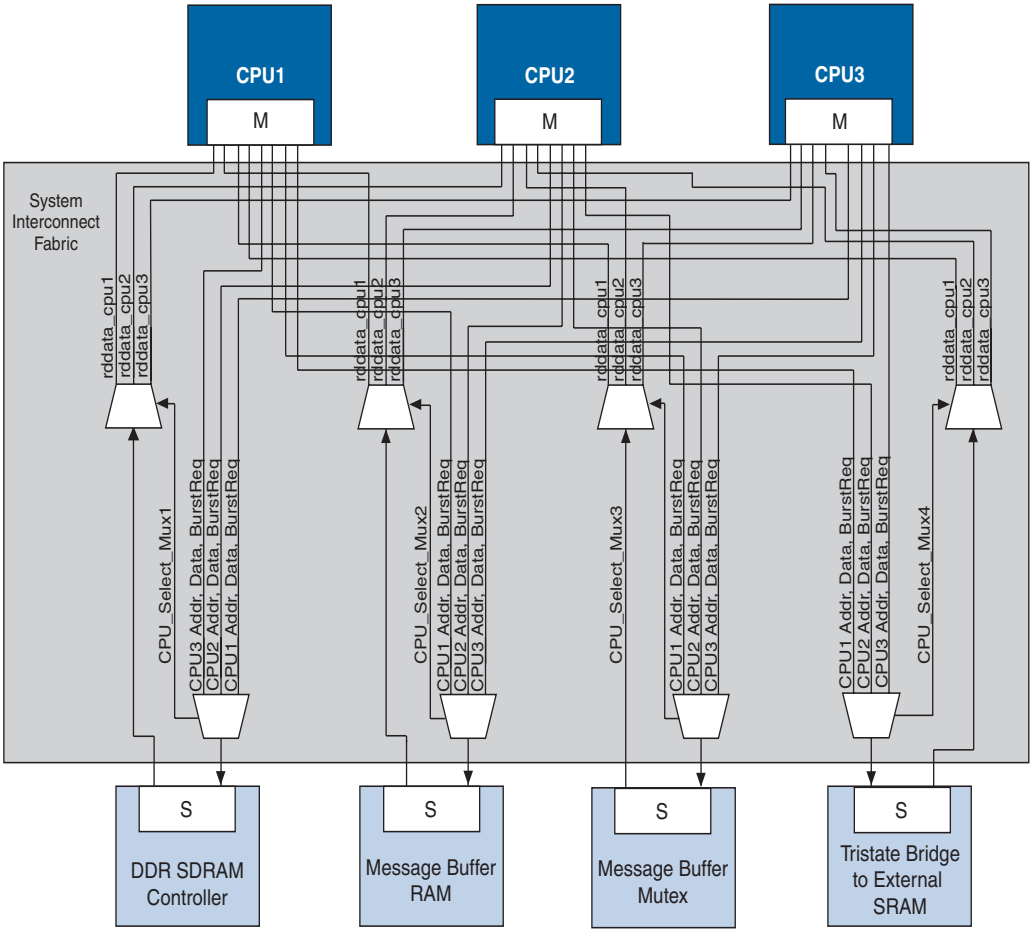


Figure 10–4 and Figure 10–5 show how you can improve the logic utilization of the system interconnect fabric by inserting bridges. If the DDR SDRAM controller can run at 166 MHz and the CPUs accessing it can run at 120 MHz, inserting an Avalon-MM clock-crossing bridge between the CPUs and the DDR SDRAM has the following benefits:

- Allows the CPU and DDR interfaces to run at different frequencies.
- Places system interconnect fabric for the arbitration logic and multiplexer for the DDR SDRAM controller in the slower clock domain.

- Reduces the complexity of the interconnect logic in the faster domain, allowing the system to achieve a higher f_{MAX}.

In the system illustrated in Figure 10–4 the message buffer RAM and message buffer mutex must respond quickly to the CPUs, but each response includes only a small amount of data. Placing an Avalon-MM pipeline bridge between the CPUs and the message buffers results in the following benefits:

- Eliminates separate arbiter logic for the message buffer RAM and message buffer mutex, which reduces logic utilization and propagation delay, thus increasing the f_{MAX}.
- Reduces the overall size and complexity of the system interconnect fabric.

Figure 10–4. Example SOPC System with Bridges - SOPC Builder View

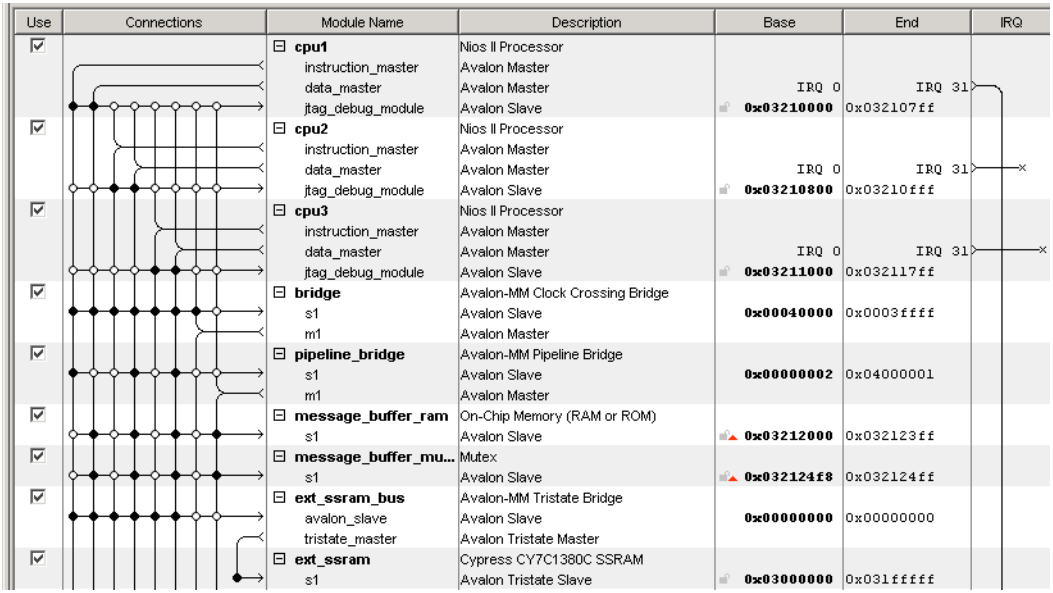
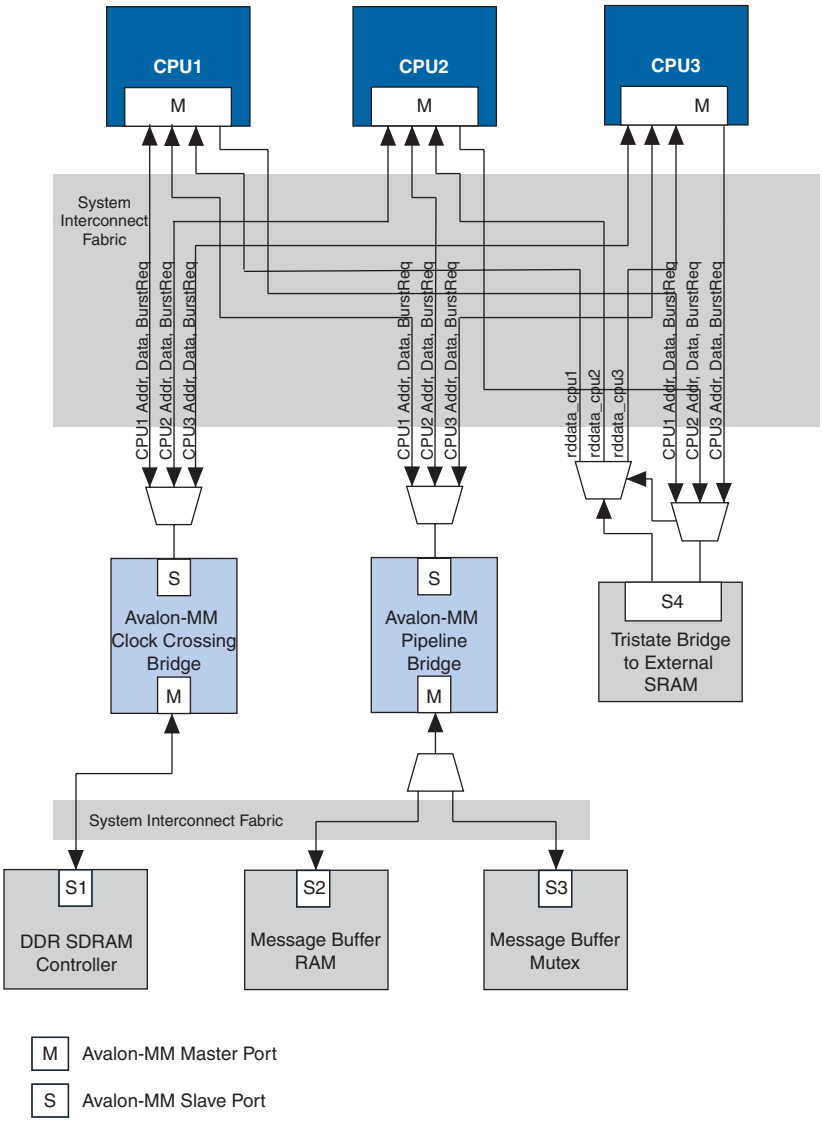


Figure 10–5 shows the system interconnect fabric that SOPC Builder would create for the system in Figure 10–4. Figure 10–5 is the same system that is pictured in Figure 10–3 except that it includes bridges to control system topology.

Figure 10–5. Example System with a Bridge



Address Mapping for Systems with Avalon-MM Bridges

An Avalon-MM bridge has an address span and range which are defined as follows:

- The address *span* of an Avalon-MM bridge is the smallest power-of-two size that encompasses all of its slave's ranges.
- The address *range* of an Avalon-MM bridge is a numerical range from its base address to its base address plus its (span -1)

(1)

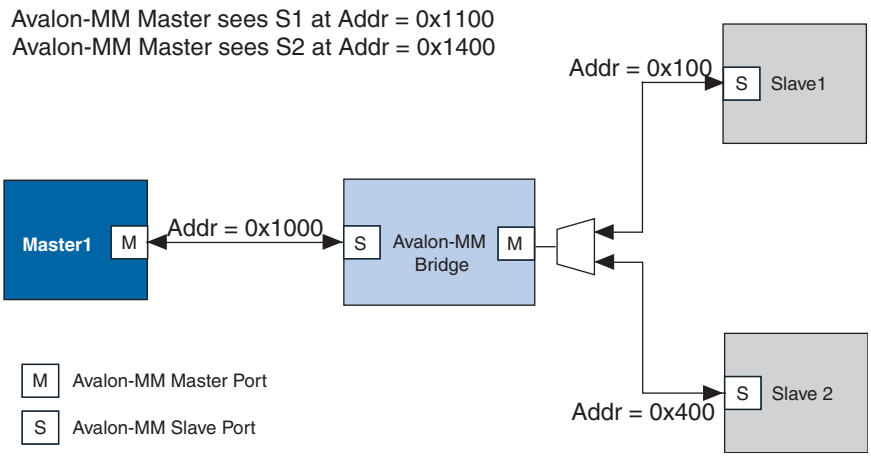
$$range = [base_address .. (base_address + (span -)];$$

SOPC Builder follows several rules in constructing an address map for a system with Avalon-MM bridges:

1. The address span of each Avalon-MM slave is rounded up to the nearest power of two.

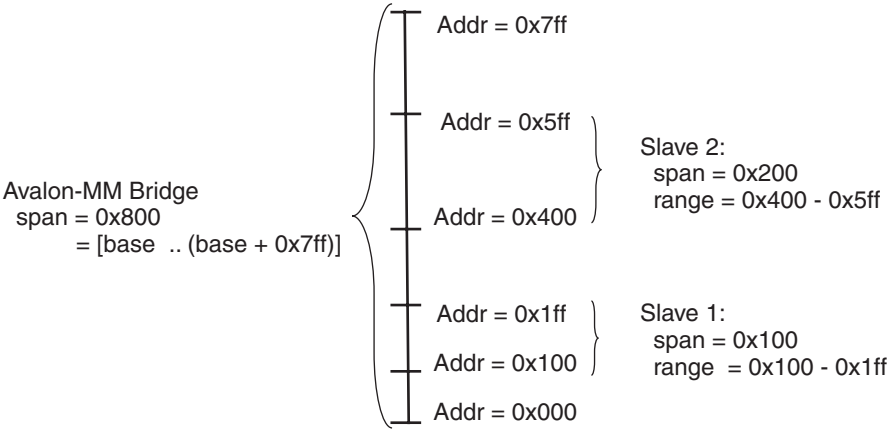
2. Each Avalon-MM slave connected to a bridge is assigned an address relative to the base address of the bridge. This address must be a multiple of its span. (See Figure 10–6.)

Figure 10–6. Avalon-MM Master and Slave Addresses



3. In the example shown in Figure 10–6, if the address span of Slave 1 is 0×100 and the address span of Slave 2 is 0×200, Figure 10–7 illustrates the address span of the Avalon-MM bridge.

Figure 10–7. The Address Span of an Avalon-MM Bridge



Tools for Visualizing the Address Map

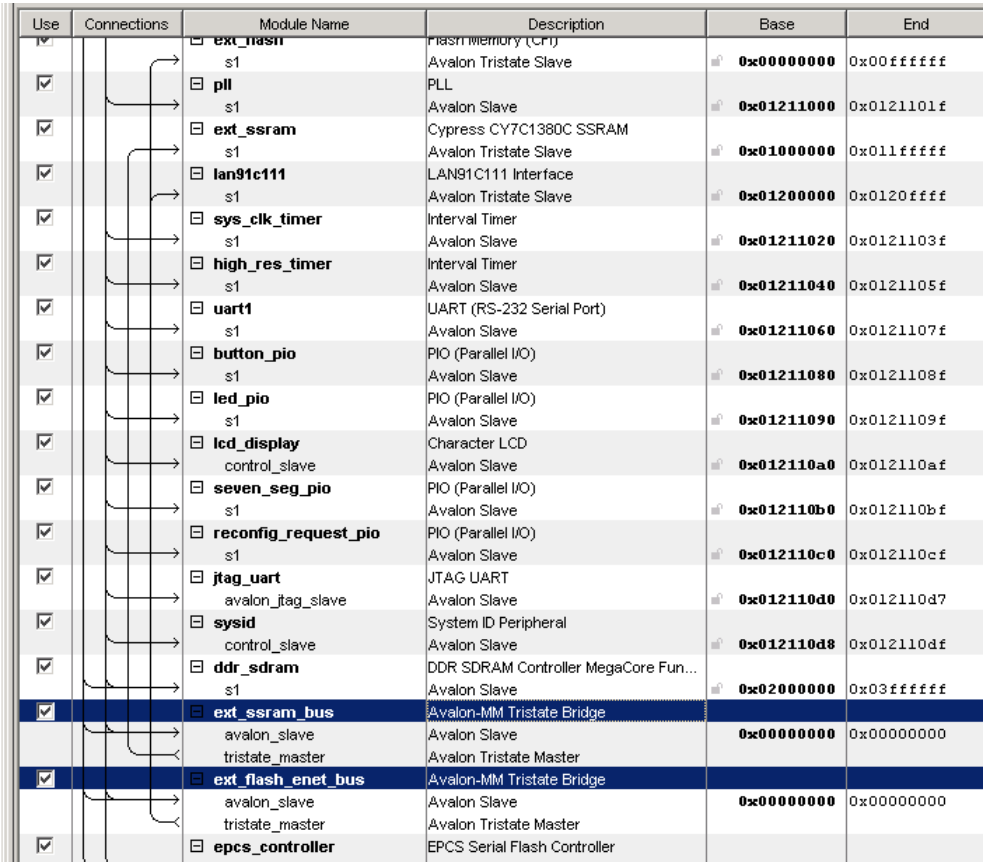
The Base Address column of SOPC Builder displays the base address *offset* of the Avalon-MM slave relative to the base address of the Avalon-MM bridge to which it is connected. You can see the absolute address map for each master in the system by clicking the **Address Map** button on the **System Components** tab.

Differences between Avalon-MM Bridges and Avalon-MM Tristate Bridges

You use Avalon-MM bridges to control topology and separate clock domains for on-chip components. You use tristate bridges to connect to off-chip components and to share pins, decreasing the overall pin count of the device. Tristate bridges are also used to change bi-directional input data into uni-directional input and output data signals. Tristate bridges are *transparent*, meaning that they do not affect the addresses of the components they connect to. All tristate bridges in a system have an address of 0×00000000 as Figure 10–8 illustrates.

For more information about the Avalon-MM tristate bridge, refer to the *Building Memory Subsystems Using SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.

Figure 10–8. SOPC Builder System with Two Tristate Bridges



Avalon-MM Pipeline Bridge

This section describes the hardware structure and functionality of the Avalon-MM pipeline bridge component.

Component Overview

The Avalon-MM pipeline bridge inserts registers in the path between its master and slave ports. In a given SOPC Builder system, if the critical register-to-register propagation delay occurs in the system interconnect fabric, the pipeline bridge can help reduce this delay and improve system f_{MAX}.

The bridge allows you to independently pipeline different groups of signals that can create a critical timing path in the interconnect:

- Master-to-slave signals, such as address, write data, and control signals
- Slave-to-master signals, such as read data
- The `waitrequest` signal to the master



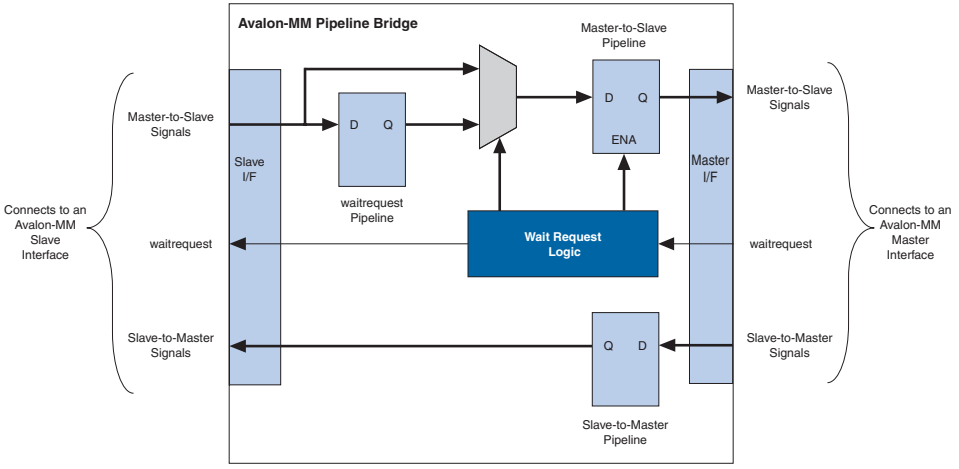
The Avalon-MM pipeline bridge can also be used to control topology without adding a pipeline stage. In this case, the pipeline bridge controls the wiring of the system interconnect fabric without adding any latency. To instantiate a bridge that does not add any pipeline stages, simply do not select any of the **Pipeline Options** on the parameter page. For the system illustrated in Figure 10–5, a pipeline bridge that does not add a pipeline register stage is optimal because the CPUs require minimal delay from the message buffer mutex and message buffer RAM. There is one instance where a pipeline bridge that does not add any register stages will fail: If a slave does not have read latency, it cannot be connected to a bridge with no pipeline stages.

The Avalon-MM pipeline bridge component is SOPC Builder-ready and integrates easily into any SOPC Builder system.

Functional Description

Figure 10–9 shows a block diagram of the Avalon-MM pipeline bridge component.

Figure 10–9. Avalon-MM Pipeline Bridge Block Diagram



The following sections describe the component's hardware functionality.

Interfaces

The bridge interface is composed of an Avalon-MM slave port and an Avalon-MM master port. The data width of the ports is configurable, which can affect how SOPC Builder generates dynamic bus sizing logic in the system interconnect fabric. Both ports support Avalon-MM pipelined transfers with variable latency. Both ports optionally support bursts of user-configurable length.

Pipeline Stages and Effects on Latency

The bridge provides three optional register stages to pipeline the following groups of signals.

- Master-to-slave signals, including:
  - `address`
  - `writedata`
  - `write`
  - `read`
  - `byteenable`
  - `chipselect`
  - `burstcount` (optional)

- Slave-to-master signals, including:
  - `readdata`
  - `readdatavalid`
  - `endofpacket`
- The `waitrequest` signal to the master port

Including a register stage affects the timing and latency of transfers through the bridge, as follows:

- Including the register stages increases latency by one cycle in each direction, but also increases the f_{MAX} by reducing propagation delay.
- Write transfers from the Avalon-MM master to the slave interface of the bridge are decoupled from write transfers from the master interface of the bridge to the slave peripheral because Avalon-MM write transfers do not require an acknowledge from the slave.
- Including the waitrequest register stage increases the latency of master-to-slave signals by one cycle for each cycle in which the `waitrequest` signal is asserted.

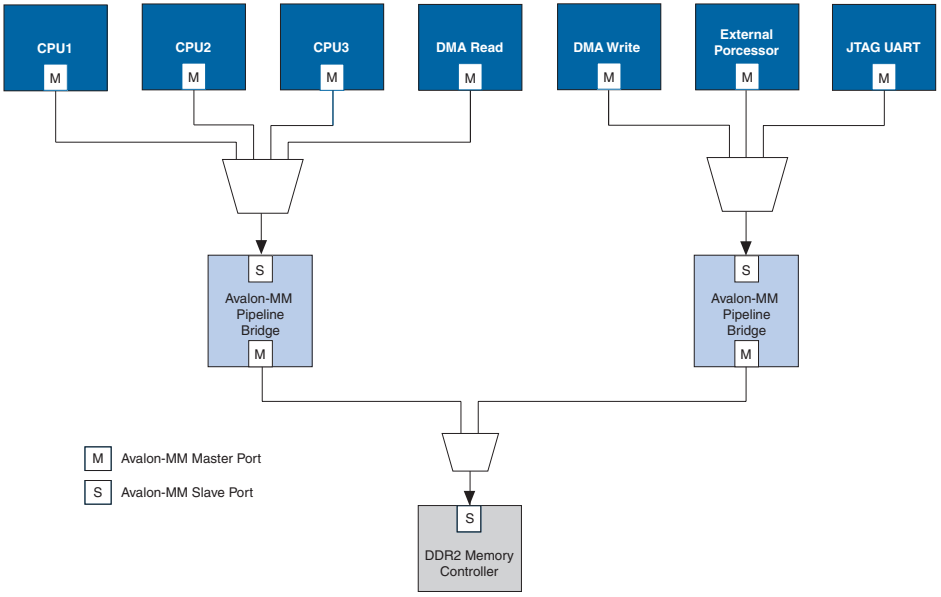
Burst Support

The bridge can optionally support bursts with configurable maximum burst length. When configured to support bursts, the bridge propagates bursts between master-slave pairs, up to the maximum burst length. Not having burst support is equivalent to a maximum burst length of one. In this case, the system interconnect fabric automatically decomposes master-to-bridge bursts into a sequence of individual transfers.

Example System with Avalon-MM Pipeline Bridges

Figure 10–10 illustrates a system in which 7 Avalon-MM masters are accessing a single DDR2 memory controller. By inserting two Avalon-MM pipeline bridges, you can limit the complexity of the multiplexer that would be required without the intermediate pipeline stage.

Figure 10–10. Seven Avalon-MM Masters Accessing One Avalon-MM Slave



Instantiating the Avalon-MM Pipeline Bridge in SOPC Builder

You use the Avalon-MM Pipeline Bridge MegaWizard interface in SOPC Builder to specify the hardware features. Refer to the *Building Memory Subsystems Using SOPC Builder* chapter in volume 4 of the *Quartus II Handbook* for a description of the options available on the **Parameter Settings** page of the configuration wizard.

Device Support

Altera device support for the bridge components is listed in Table 10–1. For each device family, a component provides either full or preliminary support:

- *Full support* means the component meets all functional and timing requirements for the device family and may be used in production designs.
- *Preliminary support* means the component meets all functional requirements, but might still be undergoing timing analysis for the device family; it can be used in production designs with caution.

Table 10–1. Device Family Support

Device Family	Avalon-MM Pipeline Bridge Support	Avalon-MM Clock-Crossing Bridge Support
Arria™ GX	Full	Preliminary
Stratix® III	Full	Preliminary
Stratix II GX	Full	Full
Stratix II	Full	Full
Stratix®	Full	Full
Cyclone™ III	Full	Preliminary
Cyclone II	Full	Full
Cyclone	Full	Full
HardCopy® II	Full	Full
MAX®	No support	No support
MAX II	Full	No support

Installation and Licensing

The bridge components are included in the Altera MegaCore® IP Library, which is an optional part of the Quartus® II software installation. After you install the MegaCore IP Library, SOPC Builder recognizes the bridge components and can instantiate them into a system.

You can use the bridge components for free without a license in any design targeting an Altera device.

Hardware Simulation Considerations

The bridge components do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

The bridge components do not have any user-visible control or status registers. Therefore, software cannot control or configure any aspect of the bridges during run-time. The bridges cannot generate interrupts.

Referenced Documents

This chapter references the following documents:

- *Avalon Memory-Mapped Interface Specification*
- *Building Memory Subsystems Using SOPC Builder*

Document Revision History

Table 10–2 shows the revision history for this chapter.

Table 10–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Moved discussion of clock-crossing bridge from this chapter to chapter 2.	—
May 2007, v7.1.0	Initial release of the document.	The Avalon-MM Pipeline Bridge and Avalon-MM Clock-Crossing Bridge are new components provided in the Quartus II software v7.1 release.

11. Avalon Streaming Interconnect Components

QII54021-7.2.0

Introduction to Interconnect Components

Avalon® Streaming (Avalon-ST) interconnect components facilitate the design of high-speed, low-latency datapaths for the system-on-a-programmable-chip (SOPC) environment. Interconnect components, in the context of SOPC Builder, are components that act as a part of the system interconnect fabric. They are not end points, but adapters that allow you to connect different, but compatible, streaming interfaces. The Avalon-ST interconnect components are typically used to connect cores that send and receive high-bandwidth data, including multiplexed streams, packets, cells, time division multiplexed (TDM) frames, and digital signal processor (DSP) data.

The interconnect components that you add to an SOPC Builder system insert logic between a source and sink interface, enabling that interface to operate correctly. This chapter describes three Avalon-ST interconnect components, also called adapters:

- "Timing Adapter" on page 11–3—adapts between source and sink interfaces that do support the `ready` signal and those that do not.
- "Data Format Adapter" on page 11–6—adapts source and sink interfaces that have different data widths.
- "Channel Adapter" on page 11–10—adapts source and sink interfaces that have different settings for the `channel` signal.

All of these interconnect components adapt initially incompatible Avalon-ST source and sink interfaces so that they function correctly, facilitating the development of high-speed, low-latency datapaths.

Interconnect Component Usage

Interconnect components can adapt the data or control signals of the Avalon-ST interface. Typical adaptations to control signals include:

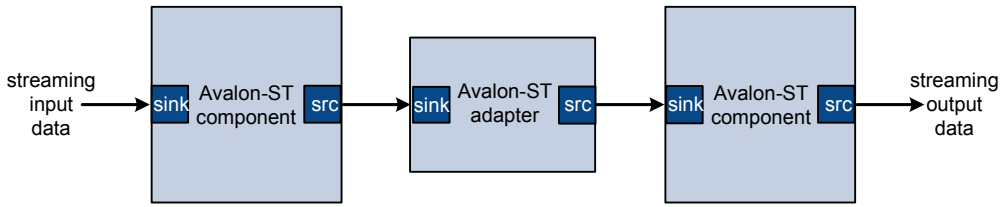
- Adding pipeline stages to adjust the timing of the `ready` signal
- Tying signals that are not used by either the source or sink to `0` or `1`

Typical adaptations to data signals include:

- Changing the number of symbols (words) that are driven per cycle
- Changing the number of channels driven

When the interconnect component adapts the data interface, it has one Avalon-ST sink interface and one Avalon-ST source interface, as shown in Figure 11–1. You configure the adapter components manually, using SOPC Builder. In contrast to the Avalon-MM interface, which allows you to create various topologies with a number of different master and slave components, the Avalon-ST interconnect components are always used to adapt point-to-point connections between streaming cores.

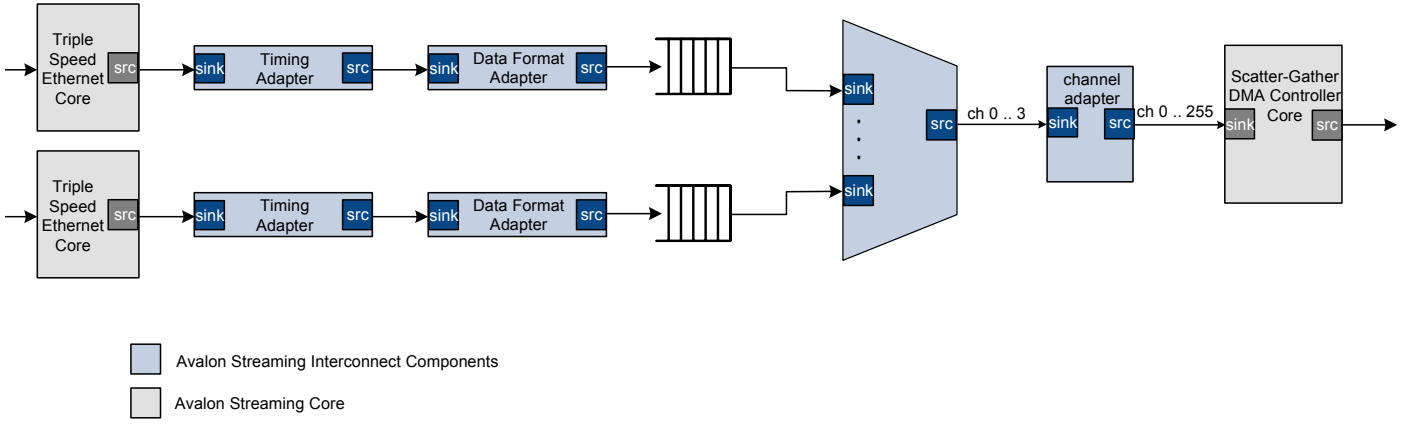
Figure 11–1. Example of an Avalon-ST Interconnect Component in an SOPC Builder System



For details about the system interconnect fabric, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*. For details about the Avalon-ST interface protocol, refer to *The Avalon Streaming Interface Specification*. Both are available at **www.altera.com**.

Figure 11–2 illustrates a datapath that connects a triple-speed Ethernet core to a scatter-gather DMA controller core using a timing adapter, data format adapter, and channel adapter so that the cores can interoperate.

Figure 11–2. Avalon-ST Datapath Constructed Using Avalon Streaming Interconnect Components



Address Mapping

The signals of the Avalon-ST source and sink interfaces are mapped into the global Avalon address space.

Timing Adapter

The timing adapter has two functions:

- It adapts source and sink interfaces that support the `ready` signal and those that do not.
- It adapts source and sink interfaces that have different ready latencies.

The timing adapter treats all signals other than the ready and valid signals as *payload,* and simply drives them from the source to the sink. Table 11–1 outlines the adaptations that the timing adapter provides.

Table 11–1. Timing Adapter

Condition	Adaptation
The source has ready, but the sink does not.	In this case, the source can respond to backpressure, but the sink never needs to apply it. The ready input to the source interface is connected directly to 1.
The source does not have ready, but the sink does.	The sink may apply backpressure, but the source is unable to respond to it. There is no logic that the adapter can insert that prevents data loss when the source asserts valid but the sink is not ready. The adapter provides simulation time error messages and an error indication if data is ever lost. The user is presented with a warning, and the connection is allowed.
The source and sink both support backpressure, but the sink's ready latency is greater than the source's.	The source responds to ready assertion or deassertion faster than the sink requires it. A number of pipeline stages equal to the difference in ready latency are inserted in the ready path from the sink back to the source, causing the source and the sink to see the same cycles as ready cycles.
The source and sink both support backpressure, but the sink's ready latency is less than the source's.	The source cannot respond to ready assertion or deassertion in time to satisfy the sink. A buffer whose depth is equal to the difference in ready latency is inserted to compensate for the source's inability to respond in time.

Resource Usage and Performance

Resource utilization for the timing adapter depends upon the function that it performs. Table 11–2 provides estimated resource utilization for seven different configurations of the timing adapter.

Table 11–2. Timing Adapter Estimated Resource Usage and Performance

Input Ready Latency	Output Ready Latency	Stratix®II and Stratix II GX (Approximate LEs)			Cyclone® II		Stratix (Approximate LEs)		
		f_{MAX} (MHz)	ALM Count	Mem Bits	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells	Mem Bits
1	2	500	2	0	420	2	422	1	0
1	3	500	2	0	420	3	422	2	0
1	4	500	4	0	420	4	422	3	0
1	0	500	21	80	420	183	422	20	80
2	1	456	21	80	401	188	317	21	80
3	1	456	21	80	401	188	317	21	80
4	1	456	21	80	401	188	317	21	80

Instantiating the Timing Adapter in SOPC Builder

You can use the Avalon-ST configuration wizard in SOPC Builder to specify the hardware features. This section describes the options available on the **Parameter Settings** page of the configuration wizard.

Input Interface Parameters

Support Backpressure with the Ready Signal—check this option to add the backpressure functionality to the interface. When the `ready` signal is used, the value for `READY_LATENCY` indicates the number of cycles between when the `ready` signal is asserted and when valid data is driven.

Output Interface Parameters

Support Backpressure with the Ready Signal—check this option to add the backpressure functionality to the interface. When the `ready` signal is used, the value for `READY_LATENCY` indicates the number of cycles between when the `ready` signal is asserted and when valid data is driven.

Common to Input and Output Interfaces

The following parameters define the interface characteristics that the adapters do not affect directly.

Channel Signal Width (Bits)

Set the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is eight bits. Set to `0` if channels are not used.

Max Channel

Set the maximum number of channels that the interface supports. Valid values are 0 - 255.

Bits Per Symbol

Set the number of bits per symbol.

Symbols Per Beat

Record the number of symbols per active transfer.

Include Packet Support

Check this box if the interfaces supports a packet protocol, including the `startofpacket`, `endofpacket` and `empty` signals.

Error Signal Width (Bits)

Record the width of the `error` signal. Valid values are 0–31 bits. Set to `0` if the `error` signal is not used.

Data Format Adapter

The data format adapter handles interfaces that have different definitions for the `data` signal. One of the more common adaptations that this adapter performs is bus width adaptation, such as converting a data

interface that drives two, 8-bit symbols per beat to an interface that drives four, 8-bit symbols per beat. The available data format adaptations are listed in Table 11–3.

Table 11–3. Data Format Adapter

Condition	Description of Adapter Logic
The source and sink's bits per symbol are different.	The connection cannot be made.
The source and sink have a different number of symbols per beat.	The adapter converts from the source's width to the sink's width. If the adaptation is from a wider to a narrower interface, a beat of data at the input will correspond to multiple beats of data at the output. If the input `error` signal is asserted for a single beat, it is asserted on output for multiple beats. If the adaptation is from a narrow to a wider interface, multiple input beats are required to fill a single output beat, and the output `error` is the logical OR of the input `error` signal.

Resource Usage and Performance

Resource utilization for the data format adapter depends upon the function that it performs. Table 11–4 provides estimated resource utilization for numerous configurations of the data format adapter.

Table 11–4. Data Format Adapter Estimated Resource Usage and Performance, 8 Bits per Symbol

Input Symbols per Beat	Output Symbols per Beat	Number of Channels	Packet Support	Stratix®II and Stratix II GX (Approximate LEs)			Cyclone® II			Stratix (Approximate LEs)		
				f_{MAX} (MHz)	ALM Count	Mem Bits	f_{MAX} (MHz)	Logic Cells	Memory Bits	f_{MAX} (MHz)	Logic Cells	Mem Bits
1	2	1	y	500	96	0	391	93	0	375	105	0
4	1	1	y	459	106	0	311	97	0	306	76	0
4	2	1	y	500	118	0	343	107	0	326	85	0
4	8	1	y	437	326	0	346	370	0	303	330	0
4	16	1	y	357	930	0	264	1005	0	231	806	0
1	2	188	y	321	110	15	187	137	15	209	153	15
4	1	105	y	244	125	2	148	183	2	150	137	2
4	2	105	y	277	101	2	172	134	2	173	108	2
4	8	130	y	322	255	41	175	279	41	187	262	41
4	16	30	y	268	341	106	166	563	106	153	471	106
4	1	105	n	269	107	2	177	185	2	167	99	2
4	2	54	n	290	109	1	193	203	1	176	91	1
4	3	10	n	249	149	18	189	251	16	159	217	18
4	5	222	n	281	300	40	199	381	40	182	316	40
4	6	30	n	312	184	40	201	385	40	198	241	40
4	7	139	n	253	285	56	159	416	56	161	427	56
4	8	198	n	311	281	40	190	247	40	198	257	40
4	15	160	n	259	370	121	165	733	121	149	697	121
4	16	36	n	227	255	105	391	93	0	146	491	105

Instantiating the Data Format Adapter in SOPC Builder

You can use the Avalon-ST configuration wizard in SOPC Builder to specify the hardware features. This section describes the options available on the **Parameter Settings** page of the configuration wizard.

Input Interface Parameters

Data Symbols Per Beat

Set the number of symbols transferred per active cycle.

Output Interface Parameters

Data Symbols Per Beat

Set the number of symbols transferred per active cycle. This value can be different for the input and output interfaces.

Common to Input and Output

The following parameters define the interface characteristics that the adapters do not affect directly.

Support Backpressure with the Ready Signal

This option adds the backpressure functionality to the interface. When the `ready` signal is used, the value for `READY_LATENCY` indicates the number of cycles between when the `ready` signal is asserted and when valid data is driven.

Data Bits Per Symbol

Record the number of bits per symbol. This value must be the same for the input and output interfaces.

Channel Signal Width (Bits)

Record the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is 8 bits. Set to 0 if channels are not used.

Max Channel

Record the maximum number of channels that the interface supports. Valid values are 0 – 255.

Include Packet Support

Turn this option on if the interface supports a packet protocol, including the `startofpacket`, `endofpacket`, and `empty` signals.

Error Signal Width (Bits)

Record the width of the `error` signal. Valid values are 0–31 bits. Set to `0` if the `error` signal is not used.

Channel Adapter

The channel adapter provides adaptations between interfaces that have different support for the `channel` signal or for the maximum number of channels supported. The adaptations are described in Table 11–5.

Table 11–5. Channel Adapter

Condition	Description of Adapter Logic
The source uses channels, but the sink does not.	The adapter provides a simulation error and signals an error for data for any channel from the source other than 0. A warning is provided to the user at generation time.
The sink has channel, but the source does not.	The user is presented with a warning, and the channel inputs to the sink are all tied to 0.
The source and sink both support channels, and the source's maximum number of channels is less than the sink's.	The source's channel is connected to the sink's channel unchanged. If the sink's channel signal has more bits, the higher bits are tied to 0.
The source and sink both support channels, but the source's maximum number of channels is greater than the sink's.	The source's channel is connected to the sink's channel unchanged. If the source's channel signal has more bits, the higher bits are left unconnected. The user is presented with a warning that channel information may be lost. An adapter provides a simulation error message and an error indication if the value of channel from the source is greater than the sink's maximum number of channels. In addition, the `valid` signal to the sink is deasserted so that the sink never sees data for channels that are out of range.

Resource Usage and Performance

The channel adapter uses fewer than 30 LEs. Its frequency is limited by the maximum frequency of the chosen device.

Instantiating the Channel Adapter in SOPC Builder

You can use the Avalon-ST configuration wizard in SOPC Builder to specify the hardware features. This section describes the options available on the **Parameter Settings** page of the configuration wizard.

Input Interface Parameters

Channel Signal Width (Bits)

Set the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is 8 bits. Set to 0 if channels are not used.

Max Channel

Set the maximum number of channels that the interface supports. Valid values are 0 – 255.

Output Interface Parameters

Channel Signal Width (Bits)

Record the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is 8 bits. Set to 0 if channels are not used.

Max Channel

Set the maximum number of channels that the interface supports. Valid values are 0 – 255.

Common to Input and Output Interfaces

Support Backpressure with the Ready Signal—Turn this option on to add the backpressure functionality to the interface. When the `ready` signal is used, the value for `READY_LATENCY` indicates the number of cycles between when the `ready` signal is asserted and when valid data is driven.

Data Bits Per Symbol

Set the number of bits per symbol.

Symbols Per Beat

Set the number of symbols per active cycle.

Include Packet Support

Turn this option on if the interface supports a packet protocol, including the `startofpacket`, `endofpacket` and `empty` signals.

Error Signal Width (Bits)

Set the width of the `error` signal. Valid values are 0–31 bits. Set to 0 if the `error` signal is not used.

Device Support

Altera device support for the Avalon-ST interconnect components is listed in Table 11–6. For each device family, a component provides either full or preliminary support:

- *Full support* means the component meets all functional and timing requirements for the device family and may be used in production designs.
- *Preliminary support* means the component meets all functional requirements, but might still be undergoing timing analysis for the device family; it may be used in production designs with caution.

Table 11–6. Device Family Support

Device Family	Timing Adapter	Data Format Adapter	Channel Adapter
Arria GX™	preliminary support	preliminary support	preliminary support
Stratix® III	preliminary support	preliminary support	preliminary support
Stratix II GX	preliminary support	preliminary support	preliminary support
Stratix II	preliminary support	preliminary support	preliminary support
Stratix	preliminary support	preliminary support	preliminary support
Cyclone III®	preliminary support	preliminary support	preliminary support
Cyclone II	preliminary support	preliminary support	preliminary support
Cyclone	preliminary support	preliminary support	preliminary support
Hardcopy® II	preliminary support	preliminary support	preliminary support

Installation and Licensing

The Avalon-ST interconnect components are included in the Altera MegaCore IP Library, which is an optional part of the Quartus® II software installation. After you install the MegaCore IP Library, SOPC Builder recognizes these components and can instantiate them into a system.

You can use the Avalon-ST components without a license in any design targeting an Altera device.

Hardware Simulation Considerations

The Avalon-ST interconnect components do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

The Avalon-ST interconnect components do not have any user-visible control or status registers. Therefore, software cannot control or configure any aspect of the interconnect components at run-time. These components cannot generate interrupts.

Referenced Documents

This chapter references the following documents:

- *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*
- *Avalon Streaming Interface Specification*

Document Revision History

Table 11–7 shows the revision history for this chapter.

Table 11–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007, v7.2.0	No changes to this release.	—
May 2007, v7.1.0	Initial release.	—



Quartus II Version 7.2 Handbook
Volume 5: Embedded Peripherals

101 Innovation Drive
San Jose, CA 95134
www.altera.com

QII5V5-7.2

Copyright © 2007 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

I.S. EN ISO 9001

Contents

Chapter Revision Dates ..... **xv**

About This Handbook ..... **xvii**
Introduction xvii
How to Contact Altera xvii
Typographic Conventions xviii

Section I. Memory Peripherals

Chapter 1. SDRAM Controller Core
Core Overview 1–1
Functional Description 1–2
Avalon-MM Interface 1–2
Off-Chip SDRAM Interface 1–3
Signal Timing and Electrical Characteristics 1–3
Synchronizing Clock and Data Signals 1–3
Clock Enable (CKE) Not Supported 1–4
Sharing Pins with Other Avalon-MM Tri-State Devices 1–4
Board Layout and Pinout Considerations 1–4
Performance Considerations 1–5
Open Row Management 1–5
Sharing Data and Address Pins 1–5
Hardware Design and Target FPGA 1–5
Device and Tools Support 1–6
Instantiating the Core in SOPC Builder 1–6
Memory Profile Page 1–7
Timing Page 1–8
Hardware Simulation Considerations 1–9
SDRAM Controller Simulation Model 1–9
SDRAM Memory Model 1–10
Using the Generic Memory Model 1–10
Using the SDRAM Manufacturer's Memory Model 1–10
Example Configurations 1–11
Software Programming Model 1–13
Clock, PLL and Timing Considerations 1–13
Factors Affecting SDRAM Timing 1–14
Symptoms of an Untuned PLL 1–14
Estimating the Valid Signal Window 1–14

Example Calculation 1–17
Referenced Documents 1–20
Document Revision History 1–21

Chapter 2. CompactFlash Core

Core Overview 2–1
Functional Description 2–2
Instantiating the Core in SOPC Builder 2–3
Required Connections 2–3
Device and Tools Support 2–4
Software Programming Model 2–4
HAL System Library Support 2–4
Software Files 2–5
Register Maps 2–5
Ide Registers 2–5
Ctl Registers 2–6
Cfctl Register 2–6
Idectl Register 2–7
Referenced Documents 2–7
Document Revision History 2–7

Chapter 3. Common Flash Interface Controller Core

Core Overview 3–1
Functional Description 3–2
Device and Tools Support 3–2
Instantiating the Core in SOPC Builder 3–3
Attributes Page 3–3
Presets Settings 3–3
Size Settings 3–3
Timing Page 3–3
Software Programming Model 3–4
HAL System Library Support 3–4
Limitations 3–4
Software Files 3–4
Referenced Documents 3–5
Document Revision History 3–6

Chapter 4. EPCS Device Controller Core

Core Overview 4–1
Functional Description 4–2
Avalon-MM Slave Interface and Registers 4–3
Device and Tools Support 4–4
Instantiating the Core in SOPC Builder 4–4
Software Programming Model 4–4
HAL System Library Support 4–5
Software Files 4–5
Document Revision History 4–6

Chapter 5. On-Chip FIFO Memory Core

Core Overview 5–1
Functional Description 5–1
Avalon-MM Write Slave to Avalon-MM Read Slave 5–2
Avalon-ST Sink to Avalon-ST Source 5–2
Avalon-MM Write Slave to Avalon-ST Source 5–3
Avalon-ST Sink to Avalon-MM Read Slave 5–5
Status Interfaces 5–7
Clocking Modes 5–7
Device and Tools Support 5–7
Instantiating the Core in SOPC Builder 5–7
FIFO Settings 5–7
Depth 5–7
Clock Settings 5–8
Status Port 5–8
FIFO Implementation 5–8
Interface Parameters 5–8
Input 5–8
Output 5–8
Allow Backpressure 5–8
Avalon-MM Port Settings 5–9
Avalon-ST Port Settings 5–9
Software Programming Model 5–9
HAL System Library Support 5–9
Software Files 5–10
Programming with the On-Chip FIFO Memory 5–10
Software Control 5–11
Software Example 5–14
On-Chip FIFO Memory API 5–17
Referenced Documents 5–33
Document Revision History 5–33

Chapter 6. Scatter-Gather DMA Controller Core

Core Overview 6–1
Example Systems 6–1
Resource Usage and Performance 6–3
Comparison of SG-DMA Controller Core and DMA Controller Core 6–4
Functional Description 6–5
Memory-to-Memory Configuration 6–5
Memory-to-Stream Configuration 6–7
Stream-to-Memory Configuration 6–10
Possible Sources of Errors 6–12
Detailed Description of Each Block 6–13
Descriptor Processor Block 6–13
DMA Read Block 6–13
DMA Write Block 6–13
Device Support and Tools 6–13

Instantiating the Core in SOPC Builder 6–14
Transfer Mode 6–14
Allow Unaligned Transfers 6–14
Data and Error Widths 6–14
FIFO Depth 6–14
Hardware Simulation Considerations 6–15
Software Programming Model 6–15
HAL System Library Support 6–15
Software Files 6–15
Programming with the SG-DMA Controller 6–16
Software Control 6–16
Next Descriptor Pointer 6–18
DMA Descriptors 6–19
Timeouts 6–22
SG-DMA Controller API 6–22
Document Revision History 6–33

Chapter 7. DMA Controller Core

Core Overview 7–1
Functional Description 7–1
Setting Up DMA Transactions 7–2
The Master Read and Write Ports 7–3
Addressing and Address Incrementing 7–4
Instantiating the Core in SOPC Builder 7–4
DMA Parameters (Basic) 7–5
Transfer Size 7–5
Burst Transactions 7–5
FIFO Implementation 7–5
Advanced Options 7–6
Allowed Transactions 7–6
Device and Tools Support 7–6
Software Programming Model 7–6
HAL System Library Support 7–6
ioctl() Operations 7–7
Limitations 7–7
Software Files 7–8
Register Map 7–8
status Register 7–9
readaddress Register 7–10
writeaddress Register 7–10
length Register 7–10
control Register 7–11
Interrupt Behavior 7–13
Referenced Document 7–13
Document Revision History 7–14

Section II. Communication Peripherals

Chapter 8. JTAG UART Core

Core Overview 8–1
Functional Description 8–2
Avalon Slave Interface and Registers 8–2
Read and Write FIFOs 8–3
JTAG Interface 8–3
Host-Target Connection 8–3
Device and Tools Support 8–4
Instantiating the Core in SOPC Builder 8–4
Configuration Page 8–4
Write FIFO Settings 8–5
Read FIFO Settings 8–5
Simulation Settings 8–6
Simulated Input Character Stream 8–6
Prepare Interactive Windows 8–6
Hardware Simulation Considerations 8–7
Software Programming Model 8–7
HAL System Library Support 8–7
Driver Options: Fast versus Small Implementations 8–9
ioctl() Operations 8–10
Software Files 8–11
Accessing the JTAG UART Core via a Host PC 8–11
Register Map 8–11
Data Register 8–12
Control Register 8–13
Interrupt Behavior 8–13
Referenced Document 8–14
Document Revision History 8–15

Chapter 9. UART Core

Core Overview 9–1
Functional Description 9–2
Avalon-MM Slave Interface and Registers 9–2
RS-232 Interface 9–3
Transmitter Logic 9–3
Receiver Logic 9–4
Baud Rate Generation 9–4
Device and Tools Support 9–4
Instantiating the Core in SOPC Builder 9–5
Configuration Settings 9–5
Baud Rate Options 9–5
Baud Rate (bps) Setting 9–5
Baud Rate Can Be Changed By Software Setting 9–6
Data Bits, Stop Bits, Parity 9–6

Data Bits Setting 9–6
Parity Setting 9–6
Flow Control 9–7
Include CTS/RTS Pins and Control Register Bits 9–7
Avalon-MM Transfers with Flow Control (DMA) 9–7
Include End-of-Packet Register 9–7
Simulation Settings 9–8
Simulated RXD-Input Character Stream 9–8
Prepare Interactive Windows 9–8
Create ModelSim Alias to Open Streaming Output Window 9–8
Create ModelSim Alias to Open Interactive Stimulus Window 9–8
Simulated Transmitter Baud Rate 9–9
Hardware Simulation Considerations 9–9
Software Programming Model 9–9
HAL System Library Support 9–10
Driver Options: Fast Versus Small Implementations 9–11
ioctl() Operations 9–12
Limitations 9–13
Software Files 9–13
Legacy SDK Routines 9–13
Register Map 9–14
rxdata Register 9–15
txdata Register 9–15
status Register 9–15
control Register 9–18
divisor Register (Optional) 9–19
endofpacket Register (Optional) 9–20
Interrupt Behavior 9–20
Referenced Documents 9–21
Document Revision History 9–22

Chapter 10. SPI Core

Core Overview 10–1
Functional Description 10–1
Example Configurations 10–2
Transmitter Logic 10–4
Receiver Logic 10–4
Master and Slave Modes 10–4
Master Mode Operation 10–5
Slave Mode Operation 10–6
Multi-Slave Environments 10–6
Avalon-MM Interface 10–7
Instantiating the SPI Core in SOPC Builder 10–7
Master/Slave Settings 10–7
Generate Select Signals 10–7
SPI Clock (sclk) Rate 10–8
Specify Delay 10–8

Data Register Settings 10–9
Timing Settings 10–9
Device and Tools Support 10–10
Software Programming Model 10–10
Hardware Access Routines 10–10
Software Files 10–12
Legacy SDK Routines 10–12
Register Map 10–12
rxdata Register 10–13
txdata Register 10–13
status Register 10–13
control Register 10–14
slaveselect Register 10–15
Referenced Document 10–15
Document Revision History 10–16

Section III. Display Peripherals

Chapter 11. Optrex 16207 LCD Controller Core

Core Overview 11–1
Functional Description 11–1
Device and Tools Support 11–2
Instantiating the Core in SOPC Builder 11–2
Software Programming Model 11–2
HAL System Library Support 11–2
Displaying Characters on the LCD 11–3
Software Files 11–4
Register Map 11–4
Interrupt Behavior 11–4
Referenced Document 11–4
Document Revision History 11–5

Chapter 12. Video Sync Generator and Pixel Converter Cores

Core Overview 12–1
Video Sync Generator 12–2
Functional Description 12–2
Instantiating the Core in SOPC Builder 12–3
Signals 12–3
Timing Diagrams 12–4
Pixel Converter 12–5
Functional Description 12–5
Instantiating the Core in SOPC Builder 12–6
Signals 12–6
Device and Tools Support 12–6
Hardware Simulation Considerations 12–7

Referenced Document 12–7
Document Revision History 12–7

Section IV. Multiprocessor Coordination Peripherals

Chapter 13. Mutex Core

Core Overview 13–1
Functional Description 13–1
Device and Tools Support 13–2
Instantiating the Core in SOPC Builder 13–2
Software Programming Model 13–2
Software Files 13–3
Hardware Mutex 13–3
Mutex API 13–4
Document Revision History 13–11

Chapter 14. Mailbox Core

Core Overview 14–1
Functional Description 14–1
Device and Tools Support 14–2
Instantiating the Core in SOPC Builder 14–2
Software Programming Model 14–3
Software Files 14–4
Programming with the Mailbox Core 14–4
Mailbox API 14–6
Referenced Document 14–12
Document Revision History 14–12

Section V. Other Memory-Mapped Peripherals

Chapter 15. PIO Core

Core Overview 15–1
Functional Description 15–1
Data Input and Output 15–2
Edge Capture 15–3
IRQ Generation 15–3
Example Configurations 15–4
Avalon-MM Interface 15–4
Instantiating the PIO Core in SOPC Builder 15–5
Basic Settings 15–5
Input Options 15–5
Edge Capture Register 15–5
Synchronously Capture 15–5

Enable Bit Clearing for Edge Capture Register 15–6
Interrupt 15–6
Device and Tools Support 15–6
Software Programming Model 15–6
Software Files 15–7
Legacy SDK Routines 15–7
Register Map 15–7
data Register 15–7
direction Register 15–8
interruptmask Register 15–8
edgecapture Register 15–8
Interrupt Behavior 15–9
Software Files 15–9
Referenced Document 15–9
Document Revision History 15–10

Chapter 16. Timer Core

Core Overview 16–1
Functional Description 16–2
Avalon-MM Slave Interface 16–3
Device and Tools Support 16–3
Instantiating the Core in SOPC Builder 16–3
Timeout Period 16–3
Hardware Options 16–4
Register Options 16–4
Output Signal Options 16–5
Configuring the Timer as a Watchdog Timer 16–5
Software Programming Model 16–6
HAL System Library Support 16–6
System Clock Driver 16–6
Timestamp Driver 16–6
Limitations 16–7
Software Files 16–7
Register Map 16–7
status Register 16–8
control Register 16–8
periodl and periodh Registers 16–9
snapl and snaph Registers 16–10
Interrupt Behavior 16–10
Referenced Document 16–10
Document Revision History 16–11

Chapter 17. System ID Core

Core Overview 17–1
Functional Description 17–1
Device and Tools Support 17–2
Instantiating the Core in SOPC Builder 17–2

Software Programming Model 17–3
Document Revision History 17–5

Chapter 18. PLL Core

Core Overview 18–1
Functional Description 18–2
altpll Megafunction 18–2
Clock Outputs 18–3
PLL Status and Control Signals 18–3
System Reset Considerations 18–3
Device and Tools Support 18–3
Instantiating the Core in SOPC Builder 18–4
PLL Settings Page 18–4
Interface Page 18–4
Finish 18–5
Hardware Simulation Considerations 18–6
Register Definitions and Bit List 18–6
Status Register 18–6
Control Register 18–7
Referenced Document 18–7
Document Revision History 18–8

Chapter 19. Performance Counter Core

Core Overview 19–1
Functional Description 19–2
Section Counters 19–2
Global Counter 19–2
Register Map 19–3
System Reset Considerations 19–4
Device and Tools Support 19–4
Instantiating the Core in SOPC Builder 19–4
Define Counters 19–4
Multiple Clock Domain Considerations 19–4
Hardware Simulation Considerations 19–4
Software Programming Model 19–5
Software Files 19–5
Using the Performance Counter 19–5
API Summary 19–5
Functions and macros 19–5
Hardware constants 19–6
Startup 19–6
Global Counter Usage 19–6
Section Counter Usage 19–6
Viewing Counter Values 19–7
Interrupt Behavior 19–8
Performance Counter API 19–8
Referenced Document 19–19

Document Revision History 19–19

Section VI. Streaming Peripherals

Chapter 20. Avalon Streaming Channel Multiplexer and Demultiplexer Cores

Core Overview 20–1
Resource Usage and Performance 20–2
Multiplexer 20–3
Functional Description 20–3
Input Interfaces 20–4
Output Interface 20–4
Instantiating the Multiplexer in SOPC Builder 20–5
Demultiplexer 20–6
Functional Description 20–6
Input Interface 20–6
Output Interfaces 20–6
Instantiating the Demultiplexer in SOPC Builder 20–7
Device and Tools Support 20–8
Installation and Licensing 20–9
Hardware Simulation Considerations 20–9
Software Programming Model 20–9
Document Revision History 20–9

Chapter 21. Avalon Streaming Test Pattern Generator and Checker Cores

Core Overview 21–1
Resource Utilization and Performance 21–2
Test Pattern Generator 21–4
Functional Description 21–4
Command Interface 21–4
Control and Status Interface 21–5
Output Interface 21–5
Instantiating the Test Pattern Generator in SOPC Builder 21–5
Functional Parameter 21–5
Output Interface 21–6
Test Pattern Checker 21–6
Functional Description 21–6
Input Interface 21–7
Control and Status Interface 21–7
Instantiating the Test Pattern Checker in SOPC Builder 21–8
Functional Parameter 21–8
Input Parameters 21–8
Device and Tools Support 21–8
Installation and Licensing 21–9
Hardware Simulation Considerations 21–9
Software Programming Model 21–10

HAL System Library Support 21–10
Software Files 21–10
Register Maps 21–10
Test Pattern Generator Control and Status Registers 21–11
Test Pattern Generator Command Registers 21–12
Test Pattern Checker Control and Status Registers 21–13
Test Pattern Generator API 21–15
data_source_reset() 21–15
data_source_init() 21–16
data_source_get_id() 21–16
data_source_get_supports_packets() 21–16
data_source_get_num_channels() 21–17
Test Pattern Checker API 21–21
data_sink_reset() 21–21
data_sink_init() 21–21
data_sink_get_id() 21–21
data_sink_get_supports_packets() 21–22
data_sink_get_num_channels() 21–22
Referenced Document 21–27
Document Revision History 21–27

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook, Volume 5*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1. SDRAM Controller Core
Revised: *October 2007*
Part number: *NII51005-7.2.0*

Chapter 2. CompactFlash Core
Revised: *October 2007*
Part number: *QII55005-7.2.0*

Chapter 3. Common Flash Interface Controller Core
Revised: *October 2007*
Part number: *NII51013-7.2.0*

Chapter 4. EPCS Device Controller Core
Revised: *October 2007*
Part number: *NII51012-7.2.0*

Chapter 5. On-Chip FIFO Memory Core
Revised: *October 2007*
Part number: *QII55002-7.2.0*

Chapter 6. Scatter-Gather DMA Controller Core
Revised: *October 2007*
Part number: *QII55003-7.2.0*

Chapter 7. DMA Controller Core
Revised: *October 2007*
Part number: *NII51006-7.2.0*

Chapter 8. JTAG UART Core
Revised: *October 2007*
Part number: *NII51009-7.2.0*

Chapter 9. UART Core
Revised: *October 2007*
Part number: *NII51010-7.2.0*

Chapter 10. SPI Core
Revised: *October 2007*
Part number: *NII51011-7.2.0*

Chapter 11. Optrex 16207 LCD Controller Core
Revised: *October 2007*
Part number: *NII51019-7.2.0*

Chapter 12. Video Sync Generator and Pixel Converter Cores
Revised: *October 2007*
Part number: *QII55006-7.2.0*

Chapter 13. Mutex Core
Revised: *October 2007*
Part number: *NII51020-7.2.0*

Chapter 14. Mailbox Core
Revised: *October 2007*
Part number: *NII53001-7.2.0*

Chapter 15. PIO Core
Revised: *October 2007*
Part number: *NII51007-7.2.0*

Chapter 16. Timer Core
Revised: *October 2007*
Part number: *NII51008-7.2.0*

Chapter 17. System ID Core
Revised: *October 2007*
Part number: *NII51014-7.2.0*

Chapter 18. PLL Core
Revised: *October 2007*
Part number: *NII53002-7.2.0*

Chapter 19. Performance Counter Core
Revised: *October 2007*
Part number: *QII55001-7.2.0*

Chapter 20. Avalon Streaming Channel Multiplexer and Demultiplexer Cores
Revised: *October 2007*
Part number: *QII55004-7.2.0*

Chapter 21. Avalon Streaming Test Pattern Generator and Checker Cores
Revised: *October 2007*
Part number: *QII55007-7.2.0*

About This Handbook

Introduction

This volume describes intellectual property (IP) cores provided by Altera® for embedded systems design. These cores are installed with the Quartus® II software, and you can use them free of charge in Altera devices. Each core is SOPC Builder ready and can be instantiated in any SOPC Builder system. Most cores provide software driver support for the Altera Nios® II processor, and work seemlessly in Nios II systems.

Each chapter provides complete reference for a core, including the following information:

- Hardware structure
- Features and interface(s) to the core
- Available options when instantiating the core in SOPC Builder
- Hardware simulation considerations, if any
- Software programming model, including a description of the registers and driver functions.
- Device and tools support

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact *(1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note to table:
(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

This document uses the typographic conventions shown in the following table.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information about a particular topic.

Section I. Memory Peripherals

This section describes memory components and interfaces provided by Altera®. These components provide access to on-chip or off-chip memory for SOPC Builder systems.

See *About This Handbook* for further details.

This section includes the following chapters:

- Chapter 1, SDRAM Controller Core
- Chapter 2, CompactFlash Core
- Chapter 3, Common Flash Interface Controller Core
- Chapter 4, EPCS Device Controller Core
- Chapter 5, On-Chip FIFO Memory Core
- Chapter 6, Scatter-Gather DMA Controller Core
- Chapter 7, DMA Controller Core



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

1. SDRAM Controller Core

NII51005-7.2.0

Core Overview

The SDRAM controller core with Avalon® interface provides an Avalon Memory-Mapped (Avalon-MM) interface to off-chip SDRAM. The SDRAM controller allows designers to create custom systems in an Altera® FPGA that connect easily to SDRAM chips. The SDRAM controller supports standard SDRAM as described in the PC100 specification.

SDRAM is commonly used in cost-sensitive applications requiring large amounts of volatile memory. While SDRAM is relatively inexpensive, control logic is required to perform refresh operations, open-row management, and other delays and command sequences. The SDRAM controller connects to one or more SDRAM chips, and handles all SDRAM protocol requirements. Internal to the FPGA, the core presents an Avalon-MM slave port that appears as linear memory (that is, flat address space) to Avalon-MM master peripherals.

The core can access SDRAM subsystems with various data widths (8, 16, 32, or 64 bits), various memory sizes, and multiple chip selects. The Avalon-MM interface is latency-aware, allowing read transfers to be pipelined. The core can optionally share its address and data buses with other off-chip Avalon-MM tri-state devices. This feature is valuable in systems that have limited I/O pins, yet must connect to multiple memory chips in addition to SDRAM.

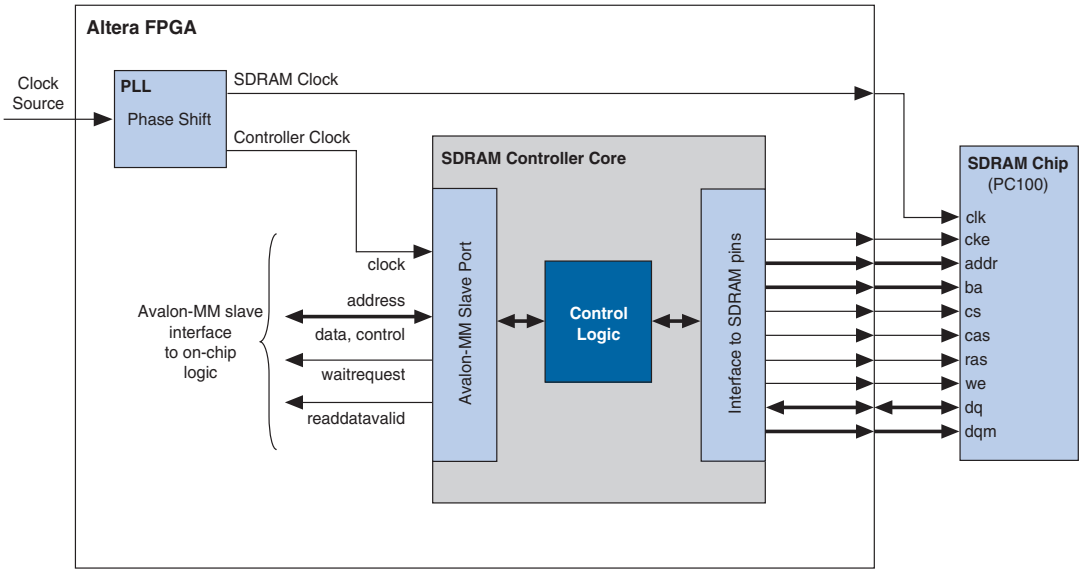
The SDRAM controller core with Avalon interface is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 1–2
- "Device and Tools Support" on page 1–6
- "Instantiating the Core in SOPC Builder" on page 1–6
- "Hardware Simulation Considerations" on page 1–9
- "Software Programming Model" on page 1–13
- "Clock, PLL and Timing Considerations" on page 1–13

Functional Description

Figure 1–1 shows a block diagram of the SDRAM controller core connected to an external SDRAM chip.

Figure 1–1. SDRAM Controller with Avalon Interface Block Diagram



The following sections describe the components of the SDRAM controller core in detail. All options are specified at system generation time, and cannot be changed at runtime.

Avalon-MM Interface

The Avalon-MM slave port is the user-visible part of the SDRAM controller core. The slave port presents a flat, contiguous memory space as large as the SDRAM chip(s). When accessing the slave port, the details of the PC100 SDRAM protocol are entirely transparent. The Avalon-MM interface behaves as a simple memory interface. There are no memory-mapped configuration registers.

The Avalon-MM slave port supports peripheral-controlled wait states for read and write transfers. The slave port stalls the transfer until it can present valid data. The slave port also supports read transfers with variable latency, enabling high-bandwidth, pipelined read transfers. When a master peripheral reads sequential addresses from the slave port, the first data returns after an initial period of latency. Subsequent reads

can produce new data every clock cycle. However, data is not guaranteed to return every clock cycle, because the SDRAM controller must pause periodically to refresh the SDRAM.



For details about Avalon-MM transfer types, refer to the *Avalon Memory-Mapped Interface Specification*.

Off-Chip SDRAM Interface

The interface to the external SDRAM chip presents the signals defined by the PC100 standard. These signals must be connected externally to the SDRAM chip(s) through I/O pins on the Altera FPGA.

Signal Timing and Electrical Characteristics

The timing and sequencing of signals depends on the configuration of the core. The hardware designer configures the core to match the SDRAM chip chosen for the system. See "Instantiating the Core in SOPC Builder" on page 1–6 for details. The electrical characteristics of the FPGA pins depend on both the target device family and the assignments made in the Quartus® II software. Some FPGA families support a wider range of electrical standards, and therefore are capable of interfacing with a greater variety of SDRAM chips. For details, see the handbook for the target FPGA family.

Synchronizing Clock and Data Signals

The clock for the SDRAM chip (hereafter "SDRAM clock") must be driven at the same frequency as the clock for the Avalon-MM interface on the SDRAM controller (hereafter "controller clock"). As in all synchronous design, you must ensure that address, data, and control signals at the SDRAM pins are stable when a clock edge arrives. As shown in Figure 1–1, you can use an on-chip phase-locked loop (PLL) to alleviate clock skew between the SDRAM controller core and the SDRAM chip. At lower clock speeds, the PLL might not be necessary. At higher clock rates, a PLL is necessary to ensure that the SDRAM clock toggles only when signals are stable on the pins. The PLL block is not part of the SDRAM controller core. If a PLL is necessary, you must instantiate it manually. You can instantiate the PLL core interface, which is an SOPC Builder component, or instantiate an altpll megafunction outside the SOPC Builder system module.

If you use a PLL, you must tune the PLL to introduce a clock phase shift so that SDRAM clock edges arrive after synchronous signals have stabilized. See "Clock, PLL and Timing Considerations" on page 1–13 for details.

For more information about instantiating a PLL in your SOPC Builder system, refer to the *PLL Core* chapter in volume 5 of the *Quartus II Handbook*. The Nios® II development tools provide example hardware designs that use the SDRAM controller core in conjunction with a PLL, which you can use as a reference for your custom designs. The Nios II development tools are available free for download from **www.altera.com**.

Clock Enable (CKE) Not Supported

The SDRAM controller does not support clock-disable modes. The SDRAM controller permanently asserts the CKE signal on the SDRAM.

Sharing Pins with Other Avalon-MM Tri-State Devices

If an Avalon-MM tri-state bridge is present in the SOPC Builder system, the SDRAM controller core can share pins with the existing tri-state bridge. In this case, the core's `addr`, `dq` (data) and `dqm` (byte-enable) pins are shared with other devices connected to the Avalon-MM tri-state bridge. This feature conserves I/O pins, which is valuable in systems that have multiple external memory chips (for example, flash, SRAM, and SDRAM), but too few pins to dedicate to the SDRAM chip. See "Performance Considerations" for details about how pin sharing affects performance.

The SDRAM addresses must connect all address bits regardless of the size of the word so that the low-order address bits on the tri-state bridge align with the low-order address bits on the memory device. It is not possible to drop A0 for memories when the smallest access size is 16 bits or A0-A1 when the smallest access size is 32 bits.

Board Layout and Pinout Considerations

When making decisions about the board layout and FPGA pinout, try to minimize the skew between the SDRAM signals. For example, when assigning the FPGA pinout, group the SDRAM signals, including the SDRAM clock output, physically close together. Also, you can use the **Fast Input Register** and **Fast Output Register** logic options in the Quartus II software. These logic options place registers for the SDRAM signals in the I/O cells. Signals driven from registers in I/O cells have similar timing characteristics, such as t_{CO}, t_{SU}, and t_{H}.

Performance Considerations

Under optimal conditions, the SDRAM controller core's bandwidth approaches one word per clock cycle. However, because of the overhead associated with refreshing the SDRAM, it is impossible to reach one word per clock cycle. Other factors affect the core's performance, as described below.

Open Row Management

SDRAM chips are arranged as multiple banks of memory, in which each bank is capable of independent open-row address management. The SDRAM controller core takes advantage of open-row management for a single bank. Continuous reads or writes within the same row and bank operate at rates approaching one word per clock. Applications that frequently access different destination banks require extra management cycles for row closings and openings.

Sharing Data and Address Pins

When the controller shares pins with other tri-state devices, average access time usually increases and bandwidth decreases. When access to the tri-state bridge is granted to other devices, the SDRAM requires row open and close overhead cycles. Furthermore, the SDRAM controller has to wait several clock cycles before it is granted access again.

To maximize bandwidth, the SDRAM controller automatically maintains control of the tri-state bridge as long as back-to-back read or write transactions continue within the same row and bank.



This behavior may degrade the average access time for other devices sharing the Avalon-MM tri-state bridge.

The SDRAM controller closes an open row whenever there is a break in back-to-back transactions, or whenever a refresh transaction is required. As a result:

- The controller cannot permanently block access to other devices sharing the tri-state bridge.
- The controller is guaranteed not to violate the SDRAM's row open time limit.

Hardware Design and Target FPGA

The target FPGA affects the maximum achievable clock frequency of a hardware design. Certain device families achieve higher f_{MAX} performance than other families. Furthermore, within a device family

faster speed grades achieve higher performance. The SDRAM controller core can achieve 100 MHz in Altera's high-performance device families, such as Stratix® series FPGAs. However, the core might not achieve 100 MHz performance in all Altera FPGA families.

The f_{MAX} performance also depends on the SOPC Builder system design. The SDRAM controller clock can also drive other logic in the system module, which might affect the maximum achievable frequency. For the SDRAM controller core to achieve f_{MAX} performance of 100 MHz, all components driven by the same clock must be designed for a 100 MHz clock rate, and timing analysis in the Quartus II software must verify that the overall hardware design is capable of 100 MHz operation.

Device and Tools Support

The SDRAM Controller with Avalon interface core supports all Altera FPGA families. Different FPGA families support different I/O standards, which may affect the ability of the core to interface to certain SDRAM chips. For details about supported I/O types, see the handbook for the target FPGA family.

Instantiating the Core in SOPC Builder

Designers use the MegaWizard® Plug-In Manager interface for the SDRAM controller in SOPC Builder to specify hardware features and simulation features. The SDRAM controller MegaWizard interface has two pages: **Memory Profile** and **Timing**. This section describes the options available on each page.

The **Presets** list offers several pre-defined SDRAM configurations as a convenience. If the SDRAM subsystem on the target board matches one of the preset configurations, you can configure the SDRAM controller core easily by selecting the appropriate preset value. The following preset configurations are defined:

- Micron MT8LSDT1664HG module
- Four SDR100 8 MByte × 16 chips
- Single Micron MT48LC2M32B2-7 chip
- Single Micron MT48LC4M32B2-7 chip
- Single NEC D4564163-A80 chip (64 MByte × 16)
- Single Alliance AS4LC1M16S1-10 chip
- Single Alliance AS4LC2M8S0-10 chip

Selecting a preset configuration automatically changes values on the **Memory Profile** and **Timing** tabs to match the specific configuration. Altering a configuration setting on any page changes the **Preset** value to **custom**.

Memory Profile Page

The **Memory Profile** page allows designers to specify the structure of the SDRAM subsystem, such as address and data bus widths, the number of chip select signals, and the number of banks. Table 1–1 lists the settings available on the **Memory Profile** page.

Table 1–1. Memory Profile Page Settings

Settings		Allowed Values	Default Values	Description
Data Width		8, 16, 32, 64	32	SDRAM data bus width. This value determines the width of the dq bus (data) and the dqm bus (byte-enable).
Architecture Settings	Chip Selects	1, 2, 4, 8	1	Number of independent chip selects in the SDRAM subsystem. By using multiple chip selects, the SDRAM controller can combine multiple SDRAM chips into one memory subsystem.
	Banks	2, 4	4	Number of SDRAM banks. This value determines the width of the ba bus (bank address) that connects to the SDRAM. The correct value is provided in the data sheet for the target SDRAM.
Address Width Settings	Row	11, 12, 13, 14	12	Number of row address bits. This value determines the width of the addr bus. The Row and Column values depend on the geometry of the chosen SDRAM. For example, an SDRAM organized as 4096 (2^{12}) rows by 512 columns has a Row value of 12.
	Column	>= 8, and less than Row value	8	Number of column address bits. For example, the SDRAM organized as 4096 rows by 512 (2^{9}) columns has a Column value of 9.
Share pins via tri-state bridge dq/dqm/addr I/O pins		checked (yes), unchecked (no)	No	When set to No, all pins are dedicated to the SDRAM chip. When set to Yes, the addr, dq, and dqm pins can be shared with a tristate bridge in the system. In this case, select the appropriate tristate bridge from the pulldown menu.
Include a functional memory model in the system testbench		Yes, No	Yes	When on, SOPC Builder creates a functional simulation model for the SDRAM chip. This default memory model accelerates the process of creating and verifying systems that use the SDRAM controller. See "Hardware Simulation Considerations" on page 1–9.

Based on the settings entered on the **Memory Profile** page, the wizard displays the expected memory capacity of the SDRAM subsystem in units of megabytes, megabits, and number of addressable words. Compare these expected values to the actual size of the chosen SDRAM to verify that the settings are correct.

Timing Page

The **Timing** page allows designers to enter the timing specifications of the SDRAM chip(s) used. The correct values are available in the manufacturer's data sheet for the target SDRAM. Table 1–2 lists the settings available on the **Timing** page.

Table 1–2. Timing Page Settings

Settings	Allowed Values	Default Value	Description
CAS latency	1, 2, 3	3	Latency (in clock cycles) from a read command to data out.
Initialization refresh cycles	1 – 8	2	This value specifies how many refresh cycles the SDRAM controller performs as part of the initialization sequence after reset.
Issue one refresh command every	—	15.625 µs	This value specifies how often the SDRAM controller refreshes the SDRAM. A typical SDRAM requires 4,096 refresh commands every 64 ms, which can be achieved by issuing one refresh command every 64 ms / 4,096 = 15.625 µs.
Delay after power up, before initialization	—	100 µs	The delay from stable clock and power to SDRAM initialization.
Duration of refresh command (`t_rfc`)	—	70 ns	Auto Refresh period.
Duration of precharge command (`t_rp`)	—	20 ns	Precharge command period.
`ACTIVE` to `READ` or `WRITE` delay (`t_rcd`)	—	20 ns	`ACTIVE` to `READ` or `WRITE` delay.
Access time (`t_ac`)	—	17 ns	Access time from clock edge. This value may depend on CAS latency.
Write recovery time (`t_wr`, No auto precharge)	—	14 ns	Write recovery if explicit precharge commands are issued. This SDRAM controller always issues explicit precharge commands.

Regardless of the exact timing values you specify, the actual timing achieved for each parameter is an integer multiple of the Avalon clock period. For the **Issue one refresh command every** parameter, the actual timing is the greatest number of clock cycles that does not exceed the

target value. For all other parameters, the actual timing is the smallest number of clock ticks that provides a value greater than or equal to the target value.

Hardware Simulation Considerations

This section discusses considerations for simulating systems with SDRAM. Three major components are required for simulation:

- A simulation model for the SDRAM controller
- A simulation model for the SDRAM chip(s), also called the memory model
- A simulation testbench that wires the memory model to the SDRAM controller pins.

Some or all of these components are generated by SOPC Builder at system generation time.

SDRAM Controller Simulation Model

The SDRAM controller design files generated by SOPC Builder are suitable for both synthesis and simulation. Some simulation features are implemented in the HDL using "translate on/off" synthesis directives that make certain sections of HDL code invisible to the synthesis tool.

The simulation features are implemented primarily for easy simulation of Nios and Nios II processor systems using the ModelSim simulator. The SDRAM controller simulation model is not ModelSim specific. However, minor changes may be required to make the model work with other simulators.



If you change the simulation directives to create a custom simulation flow, be aware that SOPC Builder overwrites existing files during system generation. Take precautions to ensure your changes are not overwritten.



For a demonstration of simulation of the SDRAM controller in the context of Nios II embedded processor systems, refer to *AN 351: Simulating Nios II Processor Designs*.

SDRAM Memory Model

This section describes the two options for simulating a memory model of the SDRAM chip(s).

Using the Generic Memory Model

If the **Include a functional memory model the system testbench** option is enabled at system generation, then SOPC Builder generates an HDL simulation model for the SDRAM memory. In the auto-generated system testbench, SOPC Builder automatically wires this memory model to the SDRAM controller pins.

Using the automatic memory model and testbench accelerates the process of creating and verifying systems that use the SDRAM controller. However, the memory model is a generic functional model that does not reflect the true timing or functionality of real SDRAM chips. The generic model is always structured as a single, monolithic block of memory. For example, even for a system that combines two SDRAM chips, the generic memory model is implemented as a single entity.

Using the SDRAM Manufacturer's Memory Model

If the **Include a functional memory model the system testbench** option is not enabled, the designer is responsible for obtaining a memory model from the SDRAM manufacturer, and manually wiring the model to the SDRAM controller pins in the system testbench.

Example Configurations

The following examples show how to connect the SDRAM controller outputs to an SDRAM chip or chips. The bus labeled `ctl` is an aggregate of the remaining signals, such as `cas_n`, `ras_n`, `cke` and `we_n`.

Figure 1–2 shows a single 128-Mbit SDRAM chip with 32-bit data. Address, data, and control signals are wired directly from the controller to the chip. The result is a 128-Mbit (16-Mbyte) memory space.

Figure 1–2. Single 128-Mbit SDRAM Chip with 32-Bit Data

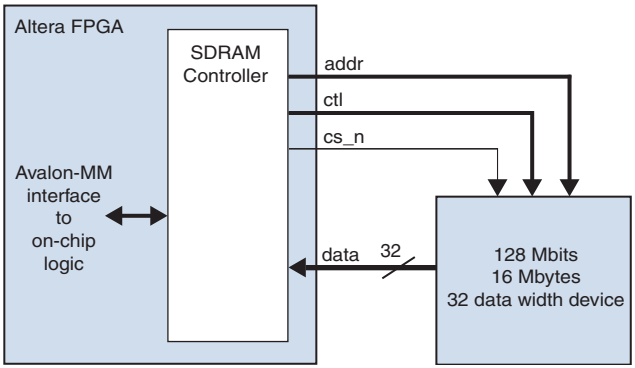


Figure 1–3 shows two 64-Mbit SDRAM chips, each with 16-bit data. Address and control signals connect in parallel to both chips. Note that chipselect (cs_n) is shared by the chips. Each chip provides half of the 32-bit data bus. The result is a logical 128-Mbit (16-Mbyte) 32-bit data memory.

Figure 1–3. Two 64-MBit SDRAM Chips Each with 16-Bit Data

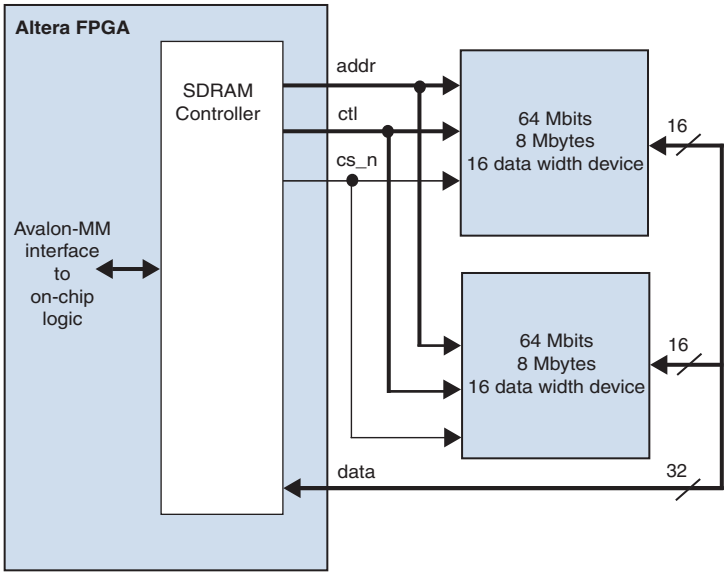
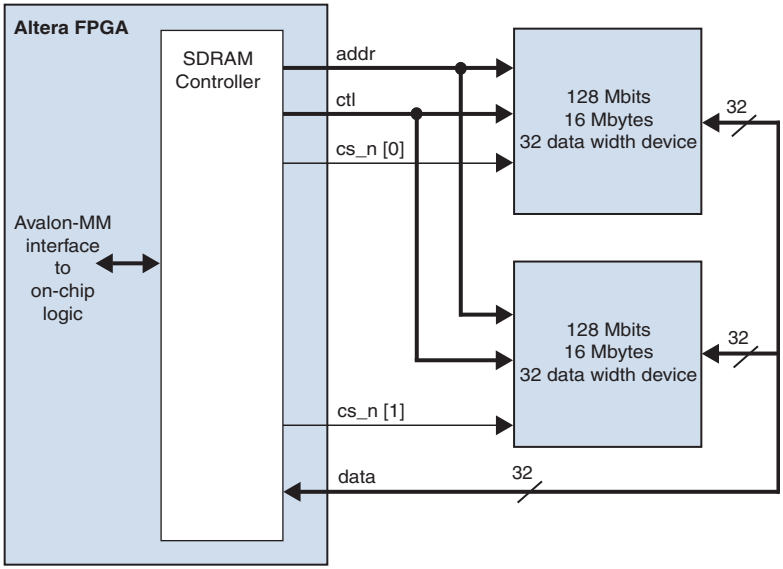


Figure 1–4 shows two 128-Mbit SDRAM chips, each with 32-bit data. Address, data, and control signals connect in parallel to the two chips. The chipselect bus (`cs_n[1:0]`) determines which chip is selected. The result is a logical 256-Mbit 32-bit wide memory.

Figure 1–4. Two 128-Mbit SDRAM Chips Each with 32-Bit Data



Software Programming Model

The SDRAM controller behaves like simple memory when accessed via the Avalon-MM interface. There are no software-configurable settings, and there are no memory-mapped registers. No software driver routines are required for a processor to access the SDRAM controller.

Clock, PLL and Timing Considerations

This section describes issues related to synchronizing signals from the SDRAM controller core with the clock that drives the SDRAM chip. During SDRAM transactions, the address, data, and control signals are valid at the SDRAM pins for a window of time, during which the SDRAM clock must toggle to capture the correct values. At slower clock frequencies, the clock naturally falls within the valid window. At higher frequencies, you must compensate the SDRAM clock to align with the valid window.

Determine when the valid window occurs either by calculation or by analyzing the SDRAM pins with an oscilloscope. Then use a PLL to adjust the phase of the SDRAM clock so that edges occur in the middle of the valid window. Tuning the PLL might require trial-and-error effort to align the phase shift to the properties of your target board.



For details about the PLL circuitry in your target device, refer to the appropriate device family handbook. For details about configuring the PLLs in Altera FPGAs, refer to the *altpll Megafunction User Guide*.

Factors Affecting SDRAM Timing

The location and duration of the window depends on several factors:

- Timing parameters of the FPGA and SDRAM I/O pins — I/O timing parameters vary based on device family and speed grade.
- Pin location on the FPGA — FPGA I/O pins connected to row routing have different timing than pins connected to column routing.
- Logic options used during the Quartus II compilation — Logic options such as the **Fast Input Register** and **Fast Output Register** logic affect the design fit. The location of logic and registers inside the FPGA affects the propagation delays of signals to the I/O pins.
- SDRAM CAS latency

As a result, the valid window timing is different for different combinations of FPGA and SDRAM devices. Furthermore, the window depends on the Quartus II software fitting results and pin assignments.

Symptoms of an Untuned PLL

Detecting when the PLL is not tuned correctly might be difficult. Data transfers to or from the SDRAM might not fail universally. For example, individual transfers to the SDRAM controller might succeed, whereas burst transfers fail. For processor-based systems, if software can perform read or write data to SDRAM, but cannot run when the code is located in SDRAM, then the PLL is probably tuned incorrectly.

Estimating the Valid Signal Window

This section describes how to estimate the location and duration of the valid signal window using timing parameters provided in the SDRAM datasheet and the Quartus II software compilation report. After finding the window, tune the PLL so that SDRAM clock edges occur exactly in the middle of the window.

Calculating the window is a two-step process. First, determine by how much time the SDRAM clock can lag the controller clock, and then by how much time it can lead. After finding the maximum lag and lead values, calculate the midpoint between them.



These calculations provide an estimation only. The following delays can also affect proper PLL tuning, but are not accounted for by these calculations.

- Signal skew due to delays on the printed circuit board — These calculations assume zero skew.
- Delay from the PLL clock output nodes to destinations — These calculations assume that the delay from the PLL SDRAM-clock output-node to the pin is the same as the delay from the PLL controller-clock output-node to the clock inputs in the SDRAM controller. If these clock delays are significantly different, you must account for this phase shift in your window calculations.

Figure 1–5 shows how to calculate the maximum length of time that the SDRAM clock can lag the controller clock, and Figure 1–6 shows how to calculate the maximum lead. Lag is a negative time shift, relative to the controller clock, and lead is a positive time shift. The SDRAM clock can lag the controller clock by the lesser of the maximum lag for a read cycle or that for a write cycle. In other words, *Maximum Lag* = minimum(*Read Lag*, *Write Lag*). Similarly, the SDRAM clock can lead by the lesser of the maximum lead for a read cycle or for a write cycle. In other words, *Maximum Lead* = minimum(*Read Lead*, *Write Lead*).

Figure 1–5. Calculating the Maximum SDRAM Clock Lag

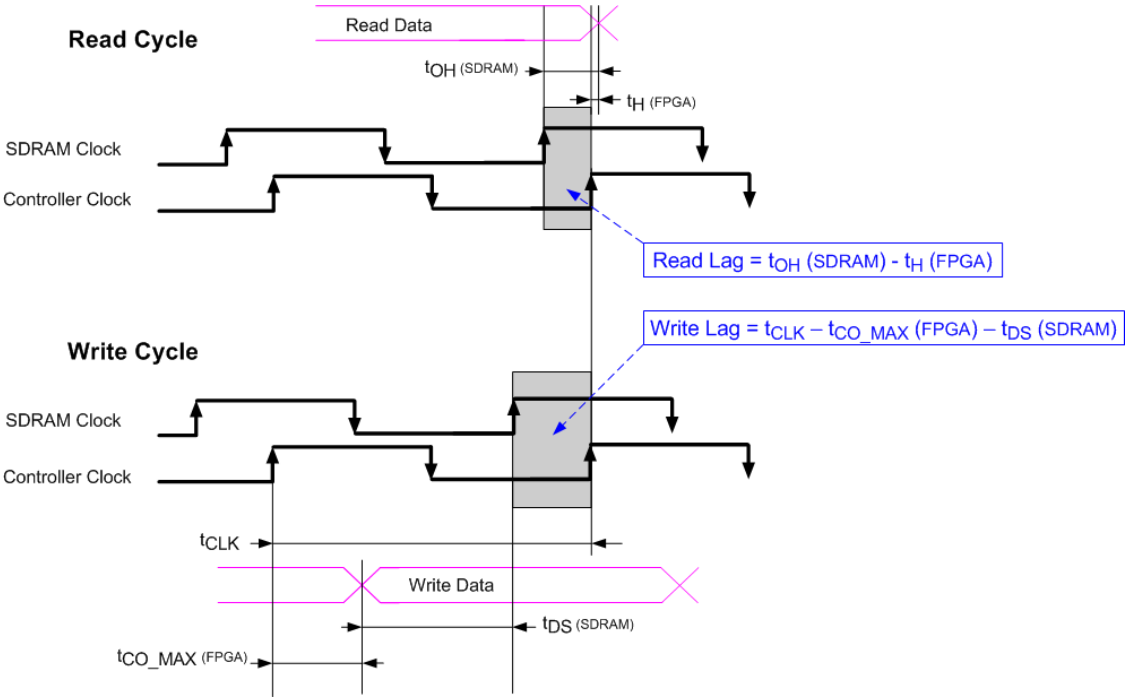
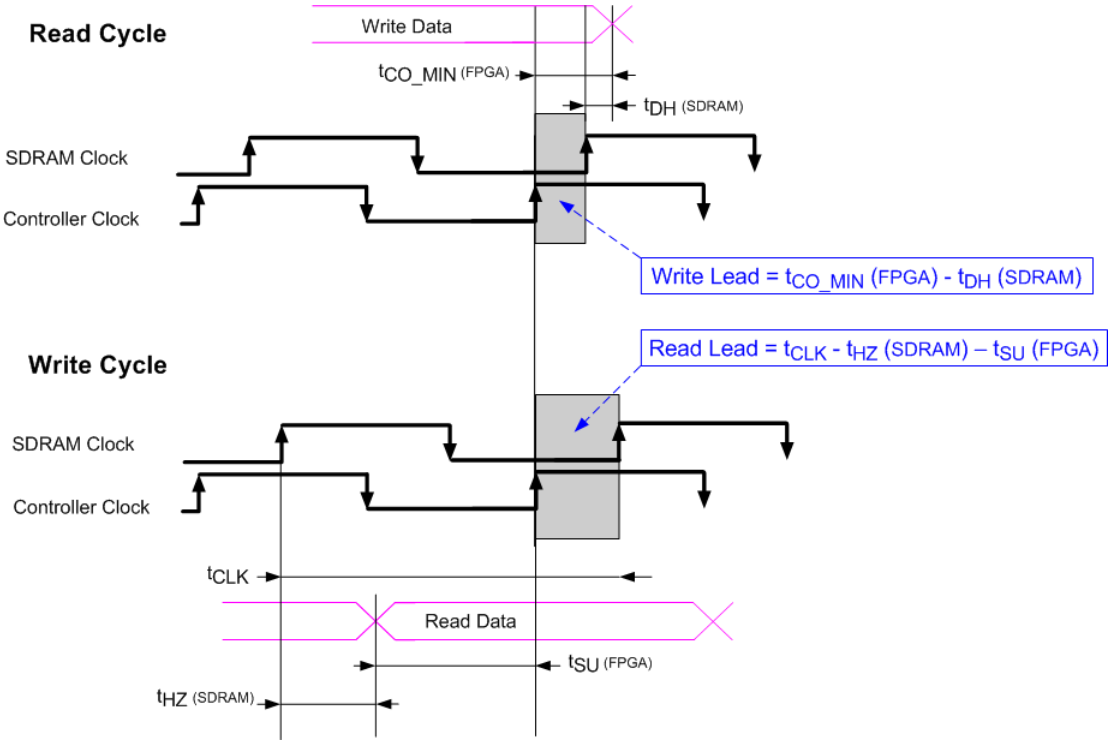


Figure 1–6. Calculating the Maximum SDRAM Clock Lead



Example Calculation

This section demonstrates a calculation of the signal window for a Micron MT48LC4M32B2-7 SDRAM chip and an FPGA design targeting an Altera Stratix II EP2S60F672C5 FPGA. This example uses a CAS latency (CL) of 3 cycles, and a clock frequency of 50 MHz. All SDRAM signals on the FPGA are registered in I/O cells, enabled with the **Fast Input Register** and **Fast Output Register** logic options in the Quartus II software.

Table 1–3 shows the relevant timing parameters excerpted from the MT48LC4M32B2 device datasheet.

Table 1–3. Timing Parameters for Micron MT48LC4M32B2 SDRAM Device

Parameter		Symbol	Value (ns) in -7 Speed Grade	
			Min.	Max.
Access time from CLK (pos. edge)	CL = 3	$t_{AC(3)}$		5.5
	CL = 2	$t_{AC(2)}$		8
	CL = 1	$t_{AC(1)}$		17
Address hold time		t_{AH}	1	
Address setup time		t_{AS}	2	
CLK high-level width		t_{CH}	2.75	
CLK low-level width		t_{CL}	2.75	
Clock cycle time	CL = 3	$t_{CK(3)}$	7	
	CL = 2	$t_{CK(2)}$	10	
	CL = 1	$t_{CK(1)}$	20	
CKE hold time		t_{CKH}	1	
CKE setup time		t_{CKS}	2	
CS#, RAS#, CAS#, WE#, DQM hold time		t_{CMH}	1	
CS#, RAS#, CAS#, WE#, DQM setup time		t_{CMS}	2	
Data-in hold time		t_{DH}	1	
Data-in setup time		t_{DS}	2	
Data-out high-impedance time	CL = 3	$t_{HZ(3)}$		5.5
	CL = 2	$t_{HZ(2)}$		8
	CL = 1	$t_{HZ(1)}$		17
Data-out low-impedance time		t_{LZ}	1	
Data-out hold time		t_{OH}	2.5	

Table 1–4 shows the relevant FPGA timing information, obtained from the Timing Analyzer section of the Quartus II Compilation Report. The values in Table 1–4 are the maximum or minimum values among all FPGA pins related to the SDRAM. The variance in timing between the SDRAM pins on the FPGA is small (less than 100 ps) because the registers for these signals are placed in the I/O cell.

Table 1–4. FPGA I/O Timing Parameters

Parameter	Symbol	Value (ns)
Clock period	tCLK	20
Minimum clock-to-output time	tCO_MIN	2.399
Maximum clock-to-output time	tCO_MAX	2.477
Maximum hold time after clock	tH_MAX	–5.607
Maximum setup time before clock	tSU_MAX	5.936



You must compile the design in the Quartus II software to obtain the I/O timing information for the FPGA design. Although Altera device family datasheets contain generic I/O timing information for each device, the Quartus II Compilation Report provides the most precise timing information for your specific design.



The timing values found in the compilation report can change, depending on fitting, pin location, and other Quartus II logic settings. When you recompile the design in the Quartus II software, verify that the I/O timing has not changed significantly.

With the values from Tables 1–3 and Table 1–4 you can perform the calculations from Figures 1–5 and 1–6, as shown below.

The SDRAM clock can lag the controller clock by the lesser of *Read Lag* or *Write Lag*:

(1) $$\text{Read Lag} = t_{OH}(\text{SDRAM}) - t_{H_MAX}(\text{FPGA})$$

$$\text{Read Lag} = 2.5\,\text{ns} - (-5.607\,\text{ns})$$

$$\text{Read Lag} = 8.107\,\text{ns}$$

or

(2) $$\text{Write Lag} = t_{CLK} - t_{CO_MAX}(\text{FPGA}) - t_{DS}(\text{SDRAM})$$

$$\text{Write Lag} = 20\,\text{ns} - 2.477\,\text{ns}$$

$$\text{Write Lag} = 15.523\,\text{ns}$$

The SDRAM clock can lead the controller clock by the lesser of *Read Lead* or *Write Lead*:

(3) $$\text{Read Lead} = t_{CO_MIN}(\text{FPGA}) - t_{DH}(\text{SDRAM})$$
$$\text{Read Lead} = 2.399\,\text{ns} - 1.0\,\text{ns}$$
$$\text{Read Lead} = 1.399\,\text{ns}$$

or

(4) $$\text{Write Lead} = t_{CLK} - t_{HZ(3)}(\text{SDRAM}) - t_{SU_MAX}(\text{FPGA})$$
$$\text{Write Lead} = 20\,\text{ns} - 5.5\,\text{ns} - 5.936\,\text{ns}$$
$$\text{Write Lead} = 8.564\,\text{ns}$$

Therefore, for this example you can shift the phase of the SDRAM clock from –8.107 ns to 1.399 ns relative to the controller clock. Choosing a phase shift in the middle of this window results in the value $(-8.107 + 1.399) \div 2 = -3.35\,\text{ns}$.

Referenced Documents

This chapter references the following documents:

- *Avalon Memory-Mapped Interface Specification*
- *PLL Core* chapter in volume 5 the *Quartus II Handbook*
- *AN 351: Simulating Nios II Processor Designs*
- *altpll Megafunction User Guide*

Document Revision History

Table 1–5 shows the revision history for this chapter.

Table 1–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No change from previous release.	—
May 2007 v7.1.0	• Updated description of Parameter Settings Memory Profile page to reflect new mechanism for for sharing pins via a tristate bridge. • Added table of contents to Overview section. • Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies. • Changed old “Avalon switch fabric” term to “system interconnect fabric.” • Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface.”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	Chapter title changed, but no change in content from previous release.	—
December 2005 v5.1.1	• Updated Figure 1-1. • Updated sections “Off-Chip SDRAM Interface” and “Board Layout and Pinout Considerations.” • Added section “Clock, PLL and Timing Considerations.”	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

2. CompactFlash Core

QII55005-7.2.0

Core Overview

The CompactFlash core allows you to connect SOPC Builder systems to CompactFlash storage cards in true IDE mode by providing an Avalon Memory-Mapped (Avalon-MM) interface to the registers on the storage cards.

The CompactFlash core also provides a register-mapped Avalon-MM slave interface which can be used by Avalon-MM master peripherals such as a Nios II processor to communicate with the CompactFlash core and manage its operations.

The CompactFlash core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

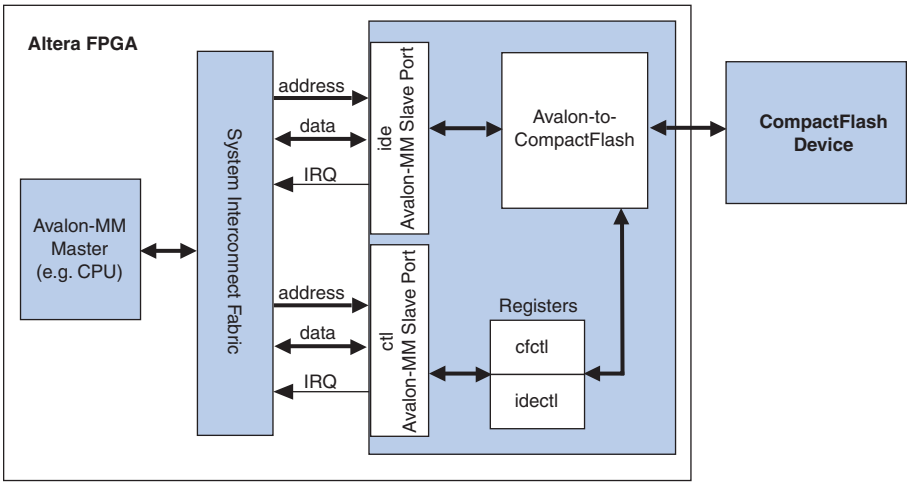
This chapter contains the following sections:

- "Functional Description" on page 2–2
- "Instantiating the Core in SOPC Builder" on page 2–3
- "Device and Tools Support" on page 2–4
- "Software Programming Model" on page 2–4

Functional Description

Figure 2–1 shows a block diagram of the CompactFlash core in a typical system configuration.

Figure 2–1. SOPC Builder System With a CompactFlash Core



As shown in Figure 2–1, the CompactFlash core provides two Avalon-MM slave interfaces: the `ide` slave port for accessing the registers on the CompactFlash device and the `ctl` slave port for accessing the core's internal registers. These registers can be used by Avalon-MM master peripherals such as a Nios II processor to control the operations of the CompactFlash core and to transfer data to and from the CompactFlash device.

You can set the CompactFlash core to generate two active-high interrupt requests (IRQs): one signals the insertion and removal of a CompactFlash device and the other passes interrupt signals from the CompactFlash device.

The CompactFlash core maps the Avalon-MM bus signals to the CompactFlash device with proper timing, thus allowing Avalon-MM master peripherals to directly access the registers on the CompactFlash device.

For more information, refer to the CF+ and CompactFlash specifications at www.compactflash.org.

Instantiating the Core in SOPC Builder

Use the MegaWizard® Plug-In Manager interface for the CompactFlash core in SOPC Builder to add the core to a system. There are no user-configurable settings for this core.

Required Connections

Table 2–1 lists the required connections between the CompactFlash core and the CompactFlash device.

Table 2–1. Required Connections (Part 1 of 2)

CompactFlash Interface Signal Name	Pin Type	CompactFlash Pin Number
addr[0]	Output	20
addr[1]	Output	19
addr[2]	Output	18
addr[3]	Output	17
addr[4]	Output	16
addr[5]	Output	15
addr[6]	Output	14
addr[7]	Output	12
addr[8]	Output	11
addr[9]	Output	10
addr[10]	Output	8
atasel_n	Output	9
cs_n[0]	Output	7
cs_n[1]	Output	32
data[0]	Input/Output	21
data[1]	Input/Output	22
data[2]	Input/Output	23
data[3]	Input/Output	2
data[4]	Input/Output	3
data[5]	Input/Output	4
data[6]	Input/Output	5
data[7]	Input/Output	6
data[8]	Input/Output	47
data[9]	Input/Output	48
data[10]	Input/Output	49

Table 2–1. Required Connections (Part 2 of 2)

CompactFlash Interface Signal Name	Pin Type	CompactFlash Pin Number
data[11]	Input/Output	27
data[12]	Input/Output	28
data[13]	Input/Output	29
data[14]	Input/Output	30
data[15]	Input/Output	31
detect	Input	25 or 26
intrq	Input	37
iord_n	Output	34
iordy	Input	42
iowr_n	Output	35
power	Output	CompactFlash power controller, if present
reset_n	Output	41
rfu	Output	44
we_n	Output	46

Device and Tools Support

The CompactFlash interface core supports all Altera FPGA families.

Software Programming Model

This section describes the software programming model for the CompactFlash core.

HAL System Library Support

The Altera-provided HAL API functions include a device driver that you can use to initialize the CompactFlash core. To perform other operations, use the low-level macros provided. For more information on the macros, refer to the files listed in the section "Software Files" on page 2–5.

Software Files

The CompactFlash core provides the following software files. These files define the low-level access to the hardware. Application developers should not modify these files.

- **altera_avalon_cf_regs.h**—The header file that defines the core's register maps.
- **altera_avalon_cf.h, altera_avalon_cf.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.

Register Maps

This section describes the register maps for the Avalon-MM slave interfaces.

Ide Registers

The `ide` port in the CompactFlash core allows you to access the IDE registers on a CompactFlash device. Table 2–2 shows the register map for the `ide` port.

Table 2–2. Ide Register Map

Offset	Register Names	
	Read Operation	Write Operation
0	RD Data	WR Data
1	Error	Features
2	Sector Count	Sector Count
3	Sector No	Sector No
4	Cylinder Low	Cylinder Low
5	Cylinder High	Cylinder High
6	Select Card/Head	Select Card/Head
7	Status	Command
14	Alt Status	Device Control

Ctl Registers

The `ctl` port in the CompactFlash core provides access to the registers which control the core's operation and interface. Table 2–3 shows the register map for the `ctl` port.

Table 2–3. Ctl Register Map

Offset	Register	Fields				
		31..4	3	2	1	0
0	`cfctl`	Reserved	`IDET`	`RST`	`PWR`	`DET`
1	`idectl`	Reserved				`IIDE`
2	Reserved	Reserved				
3	Reserved	Reserved				

Cfctl Register

The `cfctl` register controls the operations of the CompactFlash core. Reading the `cfctl` register clears the interrupt. Table 2–4 describes the `cfctl` register bits.

Table 2–4. cfctl Register Bits

Bit Number	Bit Name	Read/Write	Description
0	`DET`	RO	Detect. This bit is set to 1 when the core detects a CompactFlash device.
1	`PWR`	RW	Power. When this bit is set to 1, power is being supplied to the CompactFlash device.
2	`RST`	RW	Reset. When this bit is set to 1, the CompactFlash device is held in a reset state. Setting this bit to 0 returns the device to its active state.
3	`IDET`	RW	Detect Interrupt Enable. When this bit is set to 1, the CompactFlash core generates an interrupt each time the value of the `det` bit changes.

Idectl Register

The `idectl` register control the interface to the CompactFlash device. Table 2–5 describes the `idectl` register bit.

Table 2–5. idectl Register

Bit Number	Bit Name	Read/Write	Description
0	`IIDE`	RW	IDE Interrupt Enable. When this bit is set to 1, the CompactFlash core generates an interrupt following an interrupt generated by the CompactFlash device. Setting this bit to 0 disables the IDE interrupt.

Referenced Documents

This chapter references the *Avalon Memory-Mapped Interface Specification*.

Document Revision History

Table 2–6 shows the revision history for this chapter.

Table 2–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Initial release.	—

3. Common Flash Interface Controller Core

NII51013-7.2.0

Core Overview

The common flash interface controller core with Avalon® interface (CFI controller) allows you to easily connect SOPC Builder systems to external flash memory that complies with the Common Flash Interface (CFI) specification. The CFI controller is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

For the Nios® II processor, Altera provides hardware abstraction layer (HAL) driver routines for the CFI controller. The drivers provide universal access routines for CFI-compliant flash memories. Therefore, you do not need to write any additional code to program CFI-compliant flash devices. The HAL driver routines take advantage of the HAL generic device model for flash memory, which allows you to access the flash memory using the familiar HAL application programming interface (API) and/or the ANSI C standard library functions for file I/O. For details about how to read and write flash using the HAL API, refer to the *Nios II Software Developer's Handbook*.

The Nios II Embedded Design Suite (EDS) provides a flash programmer utility based on the Nios II processor and the CFI controller. The flash programmer utility can be used to program any CFI-compliant flash memory connected to an Altera® FPGA. For details, refer to the *Nios II Flash Programmer User Guide*.

Further information about the Common Flash Interface specification is available at **www.intel.com/design/flash/swb/cfi.htm**. As an example of a flash device supported by the CFI controller, see the data sheet for the AMD Am29LV065D-120R, available at **www.amd.com**.

The common flash interface controller core supersedes previous Altera flash cores distributed with SOPC Builder or Nios development kits. All flash chips associated with these previous cores comply with the CFI specification, and therefore are supported by the CFI controller.

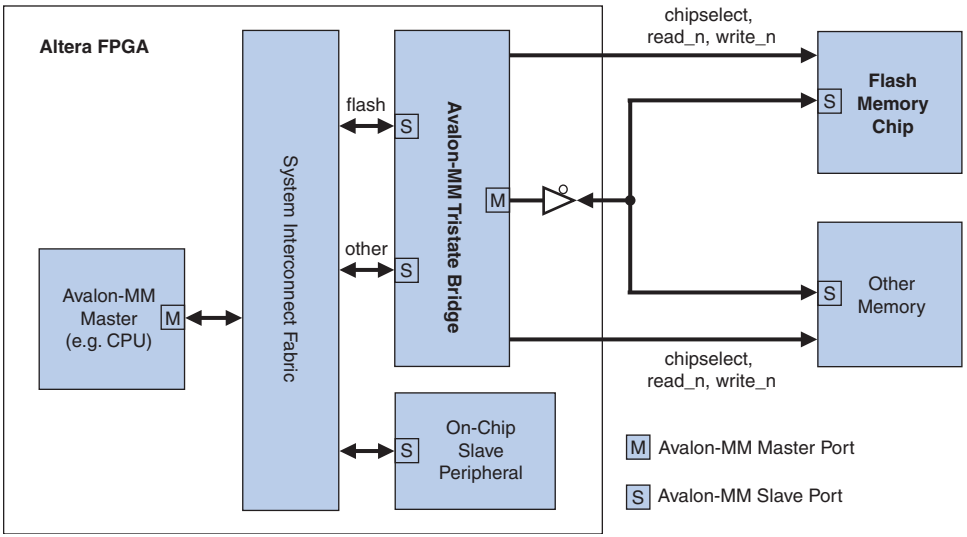
This chapter contains the following sections:

- "Functional Description" on page 3–2
- "Device and Tools Support" on page 3–2
- "Instantiating the Core in SOPC Builder" on page 3–3
- "Software Programming Model" on page 3–4

Functional Description

Figure 3–1 shows a block diagram of the CFI controller in a typical system configuration. As shown in Figure 3–1, the Avalon Memory-Mapped (Avalon-MM) interface for flash devices is connected through an Avalon-MM tristate bridge. The tristate bridge creates an off-chip memory bus that allows the flash chip to share address and data pins with other memory chips. It provides separate chipselect, read, and write pins to each chip connected to the memory bus. The CFI controller hardware is minimal: It is simply an Avalon-MM tristate slave port configured with waitstates, setup, and hold time appropriate for the target flash chip. This slave port is capable of Avalon-MM tristate slave read and write transfers.

Figure 3–1. An SOPC Builder System Integrating a CFI Controller



Avalon-MM master ports can perform read transfers directly from the CFI controller's Avalon-MM port. See "Software Programming Model" on page 3–4 for more detail on writing/erasing flash memory.

Device and Tools Support

The CFI controller supports the Arria™ GX, Stratix® III, Stratix II GX, Stratix II, Stratix GX, Stratix, Cyclone® III, Cyclone II, and Cyclone device families. The CFI controller provides drivers for the Nios II HAL system library. No software support is provided for the first-generation Nios processor.

Instantiating the Core in SOPC Builder

Hardware designers use the MegaWizard® interface for the CFI controller in SOPC Builder to specify the core features. The following sections describe the available options in the MegaWizard interface.

Attributes Page

The options on this page control the basic hardware configuration of the CFI controller.

Presets Settings

The **Presets** setting is a drop-down menu of flash chips that have already been characterized for use with the CFI controller. After you select one of the chips in the **Presets** menu, the wizard updates all settings on both tabs (except for the Board Info setting) to work with the specified flash chip.

If the flash chip on your target board does not appear in the **Presets** list, you must configure the other settings manually.

Size Settings

The size setting specifies the size of the flash device. There are two settings:

- **Address Width**—The width of the flash chip's address bus.
- **Data Width—**The width of the flash chip's data bus

The size settings cause SOPC Builder to allocate the correct amount of address space for this device. SOPC Builder will automatically generate dynamic bus sizing logic that appropriately connects the flash chip to Avalon-MM master ports of different data widths. Refer to the *Avalon Memory-Mapped Interface Specification* for details about dynamic bus sizing.

Timing Page

The options on this page specify the timing requirements for read and write transfers with the flash device. The settings available on the Timing page are:

- **Setup**—After asserting `chipselect`, the time required before asserting the `read` or `write` signals.
- **Wait**—The time required for the `read` or `write` signals to be asserted for each transfer.
- **Hold**—After deasserting the `write` signal, the time required before deasserting the `chipselect` signal.

- **Units**—The timing units used for the **Setup**, **Wait**, and **Hold** values. Possible values include ns, us, ms, and clock cycles.



For more information about signal timing for the Avalon-MM interface, refer to the *Avalon Memory-Mapped Interface Specification*.

Software Programming Model

This section describes the software programming model for the CFI controller. In general, any Avalon-MM master in the system can read the flash chip directly as a memory device. For Nios II processor users, Altera provides HAL system library drivers that enable you to erase and write the flash memory using the HAL API functions.

HAL System Library Support

The Altera-provided driver implements a HAL flash device driver that integrates into the HAL system library for Nios II systems. Programs call the familiar HAL API functions to program CFI-compliant flash memory. You do not need to know anything about the details of the underlying drivers.



The HAL API for programming flash, including C code examples, is described in detail in the *Nios II Software Developer's Handbook*. The Nios II EDS also provides a reference design called Flash Tests that demonstrates erasing, writing, and reading flash memory.

Limitations

Currently, the Altera-provided drivers for the CFI controller support only AMD and Intel flash chips.

Software Files

The CFI controller provides the following software files. These files define the low-level access to the hardware, and provide the routines for the HAL flash device driver. Application developers should not modify these files.

- **altera_avalon_cfi_flash.h, altera_avalon_cfi_flash.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.
- **altera_avalon_cfi_flash_funcs.h**, **altera_avalon_cfi_flash_table.c**—The header and source code for functions concerned with accessing the CFI table.
- **altera_avalon_cfi_flash_amd_funcs.h**, **altera_avalon_cfi_flash_amd.c**—The header and source code for programming AMD CFI-compliant flash chips.

- **altera_avalon_cfi_flash_intel_funcs.h**, **altera_avalon_cfi_flash_intel.c**—The header and source code for programming Intel CFI-compliant flash chips.

Referenced Documents

This chapter references the following documents:

- *Avalon Memory-Mapped Interface Specification*
- *Nios II Software Developer's Handbook*

Document Revision History

Table 3–1 shows the revision history for this chapter.

Table 3–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No change from previous release.	—
May 2007 v7.1.0	• Added Arria™ GX, Stratix II GX and Stratix GX to "Device and Tools Support" on page 3–2. • Removed Board Info section from MegaWizard interface because it is no longer included with the device in 7.1. • Added table of contents to Overview section. • Added Referenced Documents section.	—
March 2007 v7.0.0	Added Cyclone III support.	Version 7.0 of the Quartus II software added Cyclone III support.
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies • Changed old "Avalon switch fabric" term to "system interconnect fabric" • Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface" • Added support for Stratix III devices	For the 6.1 release, added Stratix III device support. Additionally, Altera released the Avalon Streaming interface, which necessitated some rephrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
December 2004 v1.2	Added Cyclone II support.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

4. EPCS Device Controller Core

NII51012-7.2.0

Core Overview

The EPCS device controller core with Avalon® interface allows Nios® II systems to access an Altera® EPCS serial configuration device. Altera provides drivers that integrate into the Nios II hardware abstraction layer (HAL) system library, allowing you to read and write the EPCS device using the familiar HAL application program interface (API) for flash devices.

Using the EPCS controller, Nios II systems can:

- Store program code in the EPCS device. The EPCS controller provides a boot-loader feature that allows Nios II systems to store the main program code in an EPCS device.
- Store nonvolatile program data, such as a serial number, a NIC number, and other persistent data.
- Manage the FPGA configuration data. For example, a network-enabled embedded system can receive new FPGA configuration data over a network, and use the EPCS controller to program the new data into an EPCS serial configuration device.

The EPCS controller is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. The flash programmer utility in the Nios II IDE allows you to manage and program data contents into the EPCS device.



For information about the EPCS serial configuration device family, refer to the *Serial Configuration Devices (EPCS1 & EPCS4) Data Sheet*. For details about using the Nios II HAL API to read and write flash memory, refer to the *Nios II Software Developer's Handbook*. For details about managing and programming the EPCS memory contents, refer to the *Nios II Flash Programmer User Guide*.



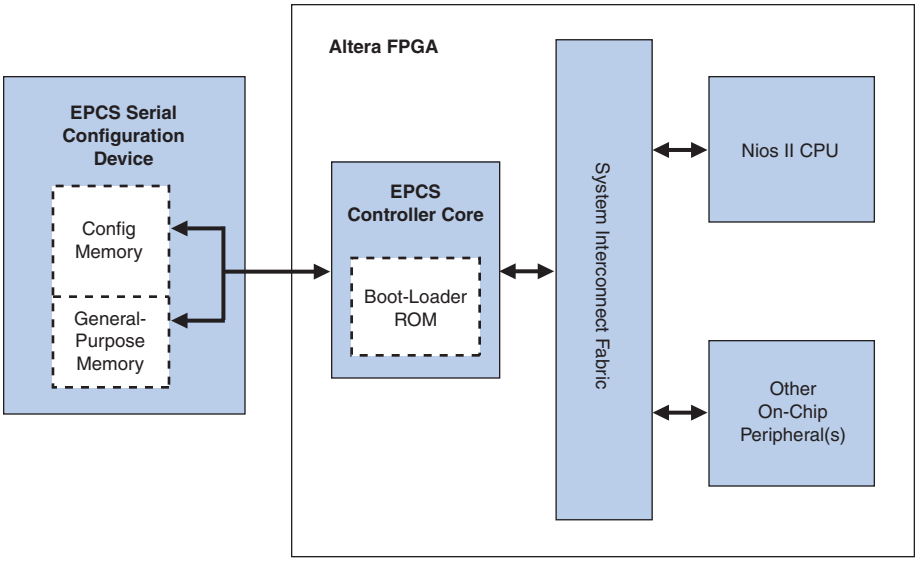
For Nios II processor users, the EPCS controller core supersedes the Active Serial Memory Interface (ASMI) device. New designs should use the EPCS controller instead of the ASMI core.

Functional Description

Figure 4–1 shows a block diagram of the EPCS controller in a typical system configuration. As shown in Figure 4–1, the EPCS device's memory can be thought of as two separate regions:

- *FPGA configuration memory*—FPGA configuration data is stored in this region.
- *General-purpose memory*—If the FPGA configuration data does not fill up the entire EPCS device, any left-over space can be used for general-purpose data and system startup code.

Figure 4–1. Nios II System Integrating an EPCS Controller



By virtue of the HAL generic device model for flash devices, accessing the EPCS device using the HAL API is the same as accessing any flash memory. The EPCS device has a special-purpose hardware interface, so Nios II programs must read and write the EPCS memory using the provided HAL flash drivers.

The EPCS controller core contains an on-chip memory for storing a boot-loader program. When used in conjunction with Cyclone®, Cyclone II, and Cyclone III devices, the core requires 512 bytes of boot-loader ROM. For Stratix® II and Stratix III devices, the core requires 1 Kbyte of boot-loader ROM. The Nios II processor can be configured to boot from the EPCS controller. To do so, set the Nios II reset address to the base address of the EPCS controller. In this case, after reset the CPU first executes code from the boot-loader ROM, which copies data from the

EPCS general-purpose memory region into a RAM. Then, program control transfers to the RAM. The Nios II IDE provides facilities to compile a program for storage in the EPCS device, and create a programming file to program into the EPCS device.



Refer to the *Nios II Flash Programmer User Guide*.

The Altera EPCS configuration device connects to the FPGA through dedicated pins on the FPGA, not through general-purpose I/O pins. In all Altera device families except Cyclone III, the EPCS controller core does not create any I/O ports on the top-level SOPC Builder system module. If the EPCS device and the FPGA are wired together on a board for configuration using the EPCS device (in other words, active serial configuration mode), no further connection is necessary between the EPCS controller and the EPCS device. When you compile the SOPC Builder system in the Quartus II software, the EPCS controller core signals are routed automatically to the device pins for the EPCS device.



If you program the EPCS device using the Quartus® II Programmer, all previous content is erased. To program the EPCS device with a combination of FPGA configuration data and Nios II program data, use the Nios II IDE flash programmer utility.

In Cyclone III, the EPCS controller does not automatically assign its output pins to the dedicated configuration pins on the FPGA. Instead, the output pins are exported to the top level design, giving users the flexibility to connect to any EPCS devices. For more information on the configuration pins in Cyclone III, refer to the Pin-Out Files for Altera Device page.

Avalon-MM Slave Interface and Registers

The EPCS controller core has a single Avalon-MM slave interface that provides access to both boot-loader code and registers that control the core. As shown in Table 4–1 on page 4–4, the first 256 words are dedicated to the boot-loader code, and the next seven words are control and data registers. A Nios II CPU can read 256 instruction words, starting from the EPCS controller's base address as flat memory space, which enables the CPU to reset into the EPCS controller's address space.

The EPCS controller core includes an interrupt signal that can be used to interrupt the CPU when a transfer has completed.

Table 4–1. EPCS Controller Register Map

Offset	Register Name	R/W	Bit Description
			31…0
0×000	Boot ROM Memory	R	Boot Loader Code
...			
0×0FF			
0×100	Read Data	R	*(1)*
0×101	Write Data	W	*(1)*
0×102	Status	R/W	*(1)*
0×103	Control	R/W	*(1)*
0×104	Reserved	—	*(1)*
0×105	Slave Enable	R/W	*(1)*
0×106	End of Packet	R/W	*(1)*

Note to Table 4–1:

(1) Altera does not publish the usage of the control and data registers. To access the EPCS device, you must use the HAL drivers provided by Altera.

Device and Tools Support

The EPCS controller supports all Altera FPGA families that support the EPCS configuration device, such as the Cyclone device family. The EPCS controller must be connected to a Nios II processor. The core provides drivers for HAL-based Nios II systems, and the precompiled boot loader code compatible with the Nios II processor. No software support is provided for any other processor, including the first-generation Nios.

Instantiating the Core in SOPC Builder

Hardware designers use the EPCS controller's SOPC Builder configuration wizard to add the EPCS controller to a system. There are no user-configurable settings for this component.

Only one EPCS controller can be instantiated in each FPGA design.

Software Programming Model

This section describes the software programming model for the EPCS controller. Altera provides HAL system library drivers that enable you to erase and write the EPCS memory using the HAL API functions. Altera does not publish the usage of the cores registers. Therefore, you must use the HAL drivers provided by Altera to access the EPCS device.

HAL System Library Support

The Altera-provided driver implements a HAL flash device driver that integrates into the HAL system library for Nios II systems. Programs call the familiar HAL API functions to program the EPCS memory. You do not need to know the details of the underlying drivers to use them.



The HAL API for programming flash, including C-code examples, is described in detail in the *Nios II Software Developer's Handbook*. For details about managing and programming the EPCS device contents, refer to the *Nios II Flash Programmer User Guide*.

Software Files

The EPCS controller provides the following software files. These files provide low-level access to the hardware and drivers that integrate into the Nios II HAL system library. Application developers should not modify these files.

- **altera_avalon_epcs_flash_controller.h**, **altera_avalon_epcs_flash_controller.c**—Header and source files that define the drivers required for integration into the HAL system library.
- **epcs_commands.h**, **epcs_commands.c**—Header and source files that directly control the EPCS device hardware to read and write the device. These files also rely on the Altera SPI core drivers.

Document Revision History

Table 4–2 shows the revision history for this chapter.

Table 4–2. Document Revision History

Date and Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Chapter 4 was formerly Chapter 3. • Added sentence stating that to boot from EPCS controller memory set Nios II reset address to the base address of the EPCS controller. • Added description on output pins assignment for Cyclone III in the Functional Description section.	—
May 2007 v7.1.0	• Removed text about reference designator from section on the configuration wizard because this setting is no longer available. • Added sentence describing the purpose of the interrupt signal.	Version 7.1 updates text for changes in the parameter sheets and to clarify use of the interrupt signal.
March 2007 v7.0.0	Added Cyclone III support.	Version 7.0 of the Quartus II software added Cyclone III support.
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies. Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface” • Changed old “Avalon switch fabric” term to “system interconnect fabric” • Added ROM memory requirements for Cyclone, Cyclone II and Stratix II devices in section “Functional Description” on page 3–2 • Added Stratix III device support	For the 6.1 release, added Stratix II support. Additionally, Altera released the Avalon Streaming interface, which necessitated some rephrasing of existing Avalon terminology. Other changes to the document serve only to clarify existing behavior.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

5. On-Chip FIFO Memory Core

QII55002-7.2.0

Core Overview

The on-chip FIFO memory core is a configurable component used to buffer data and provide flow control in an SOPC Builder system. The FIFO can operate with a single clock or with separate clocks for the input and output ports.

The input interface to the FIFO may be an Avalon® Memory Mapped (Avalon-MM) write slave or an Avalon Streaming (Avalon-ST) sink. The output interface can be a an Avalon-ST source or an Avalon-MM read slave. The data is delivered to the output interface in the same order that it was received at the input interface, regardless of the value of channel, packet, frame, or any other signals.

In single clock mode, the on-chip FIFO memory includes an optional status interface that provides information about the fill-level of the FIFO. In dual clock mode, separate, optional status interfaces can be included for the input and output interfaces. The status interface also includes registers to set and control interrupts.

The on-chip FIFO memory core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. Device drivers are provided in the HAL system library allowing software to access the core using ANSI C.

This chapter contains the following sections:

- "Functional Description"
- "Device and Tools Support" on page 5–7
- "Instantiating the Core in SOPC Builder" on page 5–7
- "Software Programming Model" on page 5–9
- "Programming with the On-Chip FIFO Memory" on page 5–10
- "On-Chip FIFO Memory API" on page 5–17

Functional Description

The on-chip FIFO memory has four configurations:

- Avalon-MM write slave to Avalon-MM read slave
- Avalon-ST sink to Avalon-ST source
- Avalon-MM write slave to Avalon-ST source
- Avalon-ST sink to Avalon-MM read slave

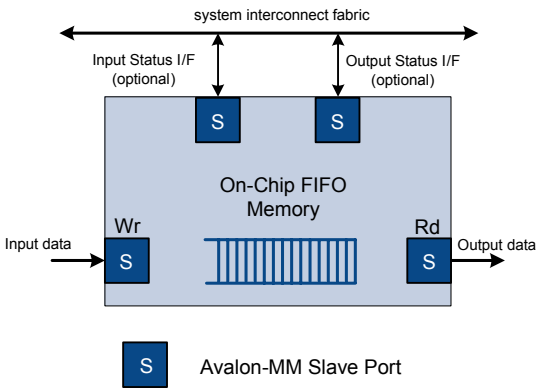
In all configurations, the input and output interfaces can use the optional backpressure signals to prevent underflow and overflow conditions. For the Avalon-MM interface, backpressure is implemented using the `waitrequest` signal. For Avalon-ST interfaces, backpressure is implemented using the `ready` and `valid` signals. For the on-chip FIFO memory, the delay between the sink asserts `ready` and the source drives valid data is one cycle. Bursting to a FIFO is not supported.

Avalon-MM Write Slave to Avalon-MM Read Slave

In this mode, the FIFO's input is a zero-address-width Avalon-MM write slave. An Avalon-MM write master pushes data into the FIFO by writing to the input interface, and a read master (possibly the same master) pops data by reading from its output interface. The FIFO's input and output data must be the same width.

If **Allow backpressure** is turned on, the `waitrequest` signal is asserted whenever the `data_in` master tries to write to a full FIFO. `waitrequest` is only deasserted when there is enough space in the FIFO for a new transaction to complete. `waitrequest` is asserted for read operations when there is no data to be read from the FIFO, and is deasserted when the FIFO has data.

Figure 5–1. FIFO with Avalon-MM Input and Output Interfaces



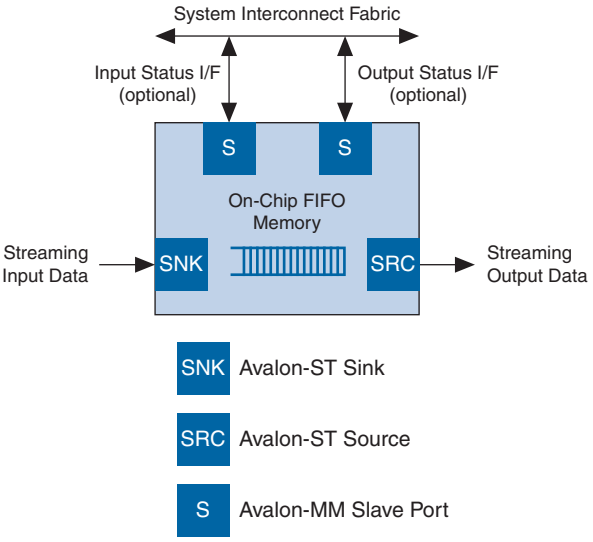
Avalon-ST Sink to Avalon-ST Source

This FIFO has streaming input and output interfaces as illustrated in Figure 5–2. You can parameterize most aspects of the Avalon-ST interfaces including the bits per symbol, symbols per beat, and the width of `error` and `channel` signals. The input and output interfaces must be

the same width. If **Allow backpressure** is on in the SOPC Builder MegaWizard, both interfaces use the `ready` and `valid` signals to indicate when space is available in the FIFO and when valid data is available.

For more information about the Avalon-ST interface protocol, refer to the *Avalon Streaming Interface Specification* available at **www.altera.com**.

Figure 5–2. FIFO with Avalon-ST Input and Output Interfaces

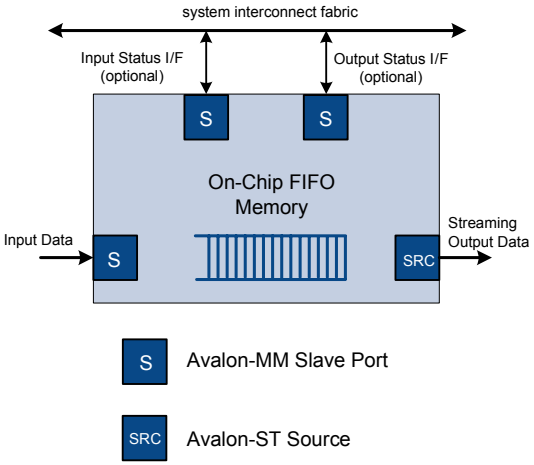


Avalon-MM Write Slave to Avalon-ST Source

In this mode, the FIFO's input is an Avalon-MM write slave with a width of 32 bits as shown in Figure 5–3. The Avalon-ST output (source) data width must also be 32 bits. You can configure output interface parameters, including: bits per symbol, symbols per beat, and the width of the `channel` and `error` signals. The FIFO performs the endian conversion to conform to the output interface protocol.

The signals that comprise this interface are mapped into bits in the Avalon's address space. If **Allow backpressure** is on, the input interface asserts `waitrequest` to indicate that the FIFO does not have enough space for the transaction to complete.

Figure 5–3. FIFO with Avalon-MM Input Interface and Avalon-ST Output Interface



The example memory map in Table 5–1 illustrates the layout of memory for a FIFO with a 32-bit Avalon-MM input interface and an Avalon-ST output interface. The output interface has 8-bit symbols, a 5-bit channel signal, and a 3-bit error signal, with packet support.

Table 5–1. Avalon-MM to Avalon-ST Memory Map

Offset	31 24	23 19	18 16	15 13	12 8	7 4	3	2	1	0
base + 0	Symbol 3	Symbol 2		Symbol 1		Symbol 0				
base + 1	reserved	reserved	error	resrvd.	channel	reserved	empty		EOP	SOP

If **Enable packet data** is off, the Avalon-MM write master writes all data at address offset 0 repeatedly to push data into the FIFO.

If **Enable packet data** is on, the Avalon-MM write master starts by writing the SOP, `error` (optional), `channel` (optional), EOP, and `empty` packet status information at address offset 1. Writing to address offset 1 does not push data into the FIFO. The Avalon-MM master then writes packet data to the FIFO repeatedly at address offset 0, pushing 8-bit symbols into the FIFO. Whenever a valid write occurs at address offset 0, the data and its respective packet information is pushed into the FIFO. Subsequent data is written at address offset 0 without the need to clear

the SOP. Rewriting to address offset 1 is not required each time if the subsequent data to be pushed into the FIFO is not the end-of-packet data, as long as `error` and `channel` do not change.

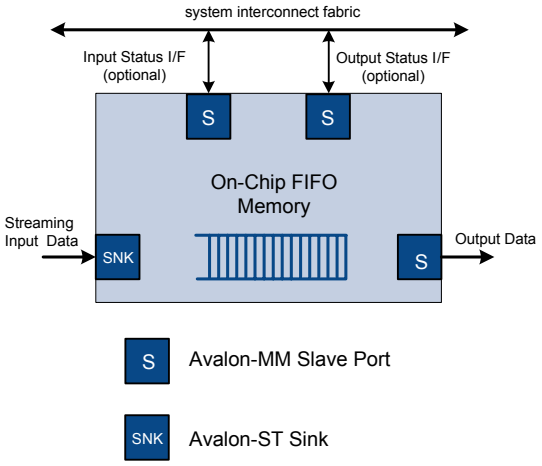
At the end of each packet, the Avalon-MM master writes to the address at offset 1 to set the EOP bit to 1, before writing the last symbol of the packet at offset 0. The write master uses the `empty` field to indicate the number of unused symbols at the end of the transfer. If the last packet data is not aligned with the symbols per beat, then the `empty` field indicates the number of empty symbols in the last packet data. For example, if the Avalon-ST interface has symbols-per-beat of 4, and the last packet only has 3 symbols, then the `empty` field will be 1, indicating that one symbol (the least significant symbol in the memory map) is empty.

Avalon-ST Sink to Avalon-MM Read Slave

In this mode, the FIFO's input is an Avalon-ST sink and the output is an Avalon-MM read slave with a width of 32 bits (Figure 5–4). The Avalon-ST input (sink) data width must also be 32 bits. You can configure input interface parameters, including: bits per symbol, symbols per beat, and the width of the `channel` and `error` signals. The FIFO performs the endian conversion to conform to the output interface protocol.

An Avalon-MM master reads the data from the FIFO. The signals are mapped into bits in the Avalon's address space. If **Allow backpressure** is on in the SOPC Builder MegaWizard, the input (sink) interface uses the `ready` and `valid` signals to indicate when space is available in the FIFO and when valid data is available. For the output interface, `waitrequest` is asserted for read operations when there is no data to be read from the FIFO. It is deasserted when the FIFO has data to send.

Figure 5–4. FIFO with Avalon-ST Input and Avalon-MM Output



As shown in Table 5–2, the memory map for the Avalon-ST to Avalon-MM slave FIFO is exactly the same as for Avalon-MM to Avalon-ST FIFO.

Table 5–2. Avalon-ST to Avalon-MM Memory Map

Offset	31 24	23 19	18 16	15 13	12 8	7 4	3	2	1	0
base + 0	Symbol 3	Symbol 2		Symbol 1		Symbol 0				
base + 1	reserved	reserved	error	resrvd.	channel	reserved	empty		EOP	SOP

If **Enable packet data** is off, read data repeatedly at address offset 0 to pop the data from the FIFO.

If **Enable packet data** is on, the Avalon-MM read master starts reading from address offset 0. If the read is valid, that is, the FIFO is not empty, both data and packet status information are popped from the FIFO. The packet status information is obtained by reading at address offset 1. Reading from address offset 1 does not pop data from the FIFO. The `error`, `channel`, SOP, EOP and `empty` fields are available at address offset 1 to determine the status of the packet data read from address offset 0.

The `empty` field indicates the number of empty symbols in the data field. For example, if the Avalon-ST interface has symbols-per-beat of 4, and the last packet data only has 1 symbol, then the `empty` field will be 3 to indicate that 3 symbols (the 3 least significant symbols in the memory map) are empty.

Status Interfaces

The FIFO provides two optional status interfaces, one for the master writing to the input interface and a second for the read master reading from the output interface. For FIFOs that operate in a single domain, a single status interface is sufficient to monitor the status of the FIFO. For FIFOs using a dual clocking scheme, a second status interface using the output clock is necessary to accurately monitor the status of the FIFO in both clock domains.

Clocking Modes

When single clock mode is used, the FIFO being used is SCFIFO. When dual-clock mode is chosen, the FIFO being used is DCFIFO. In dual-clock mode, input data and write-side status interfaces use the write side clock domain; the output data and read-side status interfaces use the read-side clock domain.

Device and Tools Support

The on-chip FIFO memory supports the Arria™ GX, Stratix® III, Stratix II GX, Stratix II, Stratix GX, Stratix, Cyclone® III, Cyclone II, Cyclone and Hardcopy® II device families.

Instantiating the Core in SOPC Builder

Designers use the MegaWizard® interface for the on-chip FIFO memory in SOPC Builder to specify the core configuration. The following sections describe the available options in the MegaWizard interface.

FIFO Settings

The following sections outline the settings that pertain to the FIFO as a whole.

Depth

Depth indicates the depth of the FIFO, in Avalon-ST beats or Avalon-MM words. The default depth is 16. When dual clock mode is used, the actual FIFO depth is equal to depth-3. This is due to clock crossing and to avoid FIFO overflow.

Clock Settings

The two options are **Single clock mode** and **Dual clock mode**. In single clock mode, all interface ports use the same clock. In dual clock mode, input data and input side status are on the input clock domain. Output data and output side status are on the output clock domain.

Status Port

The optional status ports are Avalon-MM slaves. To include the optional input side status interface, turn on **Create status interface for input** on the SOPC Builder MegaWizard. For FIFOs whose input and output ports operate in separate clock domains, you can include a second status interface by turning on **Create status interface for output**. Turning on **Enable IRQ for status ports** adds an interrupt signal to the status ports.

FIFO Implementation

This option determines if the FIFO is built from registers or embedded memory blocks. The default is to construct the FIFO from embedded memory blocks.

Interface Parameters

The following sections outline the options for the input and output interfaces.

Input

Available input interfaces are **Avalon-MM** write slave and **Avalon-ST** sink.

Output

Available output interfaces are **Avalon-MM** read slave and **Avalon-ST** source.

Allow Backpressure

When **Allow backpressure** is on, an Avalon-MM interface will include the `waitrequest` signal which is asserted to prevent a master from writing to a full FIFO or reading from an empty FIFO. An Avalon-ST interface will include the `ready` and `valid` signals to prevent underflow and overflow conditions.

Avalon-MM Port Settings

Valid **Data widths** are 8, 16, and 32 bits.

If Avalon-MM is selected for one interface and Avalon-ST for the other, the data width is fixed at 32 bits.

The Avalon-MM interface accesses data 4 bytes at a time. For data widths other than 32 bits, be cautious of potential overflow and underflow conditions.

Avalon-ST Port Settings

The following parameters allow you to specify the size and error handling of the Avalon-ST port or ports:

- **Bits per symbol**
- **Symbols per beat**
- **Channel width**
- **Error width**

If the symbol size is not a power of two, it is rounded up to the next power of two. For example, if the bits per symbol is 10, the symbol will be mapped to a 16-bit memory location. With 10-bit symbols, the maximum number of symbols per beat is two.

Enable packet data provides an option for packet transmission.

Software Programming Model

The following sections describe the software programming model for the on-chip FIFO memory core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides HAL system library drivers that enable you to access the on-chip FIFO memory core using its HAL API.

HAL System Library Support

The Altera-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access the on-chip FIFO memory via the familiar HAL API, rather than accessing the registers directly.

Software Files

Altera provides the following software files for the on-chip FIFO memory core:

- **altera_avalon_fifo_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_fifo_util.h—**This file defines functions to access the on-chip FIFO memory core hardware. It provides utilities to initialize the FIFO, read and write status, enable flags and read events.
- **altera_avalon_fifo.h—** This file provides the public interface to the on-chip FIFO memory
- **altera_avalon_fifo_util.c—**This file implements the utilities listed in **altera_avalon_fifo_util.h**.

Programming with the On-Chip FIFO Memory

This section describes the low-level software constructs for manipulating the on-chip FIFO memory core hardware. Table 5–3 lists all of the available functions.

Table 5–3. On-Chip FIFO Memory Functions (Part 1 of 2)

Function Name	Description
`altera_avalon_fifo_init()`	Initializes the FIFO.
`altera_avalon_fifo_read_status()`	Returns the integer value of the specified bit of the status register. To read all of the bits at once, use the `ALTERA_AVALON_FIFO_STATUS_ALL` mask.
`altera_avalon_fifo_read_ienable()`	Returns the value of the specified bit of the interrupt enable register. To read all of the bits at once, use the `ALTERA_AVALON_FIFO_EVENT_ALL` mask.
`altera_avalon_fifo_read_almostfull()`	Returns the value of the `almostfull` register.
`altera_avalon_fifo_read_almostempty()`	Returns the value of the `almostempty` register.
`altera_avalon_fifo_read_event()`	Returns the value of the specified bit of the event register. All of the event bits can be read at once by using the `ALTERA_AVALON_FIFO_STATUS_ALL` mask.
`altera_avalon_fifo_read_level()`	Returns the fill level of the FIFO.
`altera_avalon_fifo_clear_event()`	Clears the specified bits and the event register and performs error checking.
`altera_avalon_fifo_write_ienable()`	Writes the specified bits of the `interruptenable` register and performs error checking.

Table 5–3. On-Chip FIFO Memory Functions (Part 2 of 2)

Function Name	Description
`altera_avalon_fifo_write_almostfull()`	Writes the specified value to the `almostfull` register and performs error checking.
`altera_avalon_fifo_write_almostempty()`	Writes the specified value to the `almostempty` register and performs error checking.
`altera_avalon_fifo_write_fifo()`	Writes the specified data to the `write_address`.
`altera_avalon_fifo_write_other_info()`	Writes the packet status information to the `write_address`. Only valid when **Enable packet data** is on.
`altera_avalon_fifo_read_fifo()`	Reads data from the specified `read_address`.
`altera_avalon_fifo_read__other_info()`	Reads the packet status information from the specified `read_address`. Only valid when **Enable packet data** is on.

Software Control

Table 5–4 provides the register map for the `status` register. The layout of `status` register for the input and output interfaces is identical.

Table 5–4. FIFO Status Register Memory Map

offset	31 24	23 16	15 8	7	6	5	4	3	2	1	0
base	fill_level										
base + 1						i_status					
base + 2						event					
base + 3						interruptenable					
base + 4	almostfull										
base + 5	almostempty										

Table 5–5 outlines the use of the various fields of the `status` register.

Table 5–5. FIFO Status Field Descriptions (Part 1 of 2)

Field	Type	Description
fill_level	RO	The instantaneous fill level of the FIFO, provided in units of symbols for a FIFO with an Avalon-ST FIFO and words for an Avalon-MM FIFO.
i_status	RO	A 6-bit register that shows the FIFO's instantaneous status. See Table 5–6 for the meaning of each bit field.

Table 5–5. FIFO Status Field Descriptions (Part 2 of 2)

Field	Type	Description
event	RW1C	A 6-bit register with exactly the same fields as i_status. When a bit in the i_status register is set, the same bit in the event register is set. The bit in the event register is only cleared when software writes a 1 to that bit.
interruptenable	RW	A 6-bit interrupt enable register with exactly the same fields as the event and i_status registers. When a bit in the event register transitions from a 0 to a 1, and the corresponding bit in interruptenable is set, the master Is interrupted.
almostfull	RW	A threshold level used for interrupts and status. Can be written by the Avalon-MM status master at any time. The default threshold value for DCFIFO is Depth-4. The default threshold value for SCFIFO is Depth-1. The valid range of the threshold value is from 1 to the default. 1 will be used when attempting to write a value smaller than 1. The default will be used when attempting to write a value larger than the default.
almostempty	RW	A threshold level used for interrupts and status. Can be written by the Avalon-MM status master at any time. The default threshold value for DCFIFO is 1. The default threshold value for SCFIFO is 1. The valid range of the threshold value is from 1 to the maximum allowable almostfull threshold. 1 will be used when attempting to write a value smaller than 1. The maximum allowable will be used when attempting to write a value larger than the maximum allowable.

Table 5–6 describes the instantaneous status bits.

Table 5–6. Status Bit Field Descriptions

Bit(s)	Name	Description
1	FULL	Has a value of 1 if the FIFO is currently full.
0	EMPTY	Has a value of 1 if the FIFO is currently empty.
3	ALMOSTFULL	Has a value of 1 if the fill level of the FIFO is greater than the almostfull value.
2	ALMOSTEMPTY	Has a value of 1 if the fill level of the FIFO is less than the almostempty value.
4	OVERFLOW	Is set to 1 for 1 cycle every time the FIFO overflows. The FIFO overflows when an Avalon write master writes to a full FIFO. OVERFLOW is only valid when **Allow backpressure** is off.
5	UNDERFLOW	Is set to 1 for 1 cycle every time the FIFO underflows. The FIFO underflows when an Avalon read master reads from an empty FIFO. UNDERFLOW is only valid when **Allow backpressure** is off.

Table 5–7 lists the bit fields of the event register. These fields are identical to those in the status register and are set at the same time; however, these fields are only cleared when software writes a one to clear (W1C). The event fields can be used to determine if a particular event has occurred.

Table 5–7. Event Bit Field Descriptions

Bit(s)	Name	Description
1	E_FULL	Has a value of 1 if the FIFO has been full and the bit has not been cleared by software.
0	E_EMPTY	Has a value of 1 if the FIFO has been empty and the bit has not been cleared by software.
3	E_ALMOSTFULL	Has a value of 1 if the fill level of the FIFO has been greater than the almostfull threshold value and the bit has not been cleared by software.
2	E_ALMOSTEMPTY	Has a value of 1 if the fill level of the FIFO has been less than the almostempty value and the bit has not been cleared by software.
4	E_OVERFLOW	Has a value of 1 if the FIFO has overflowed and the bit has not been cleared by software.
5	E_UNDERFLOW	Has a value of 1 if the FIFO has underflowed and the bit has not been cleared by software.

Table 5–8 provides a mask for the six STATUS fields. When a bit in the event register transitions from a zero to a one, and the corresponding bit in the interruptenable register is set, the master is interrupted.

Table 5–8. InterruptEnable Bit Field Descriptions (Part 1 of 2)

Bit(s)	Name	Description
1	IE_FULL	Enables an interrupt if the FIFO is currently full.
0	IE_EMPTY	Enables an interrupt if the FIFO is currently empty.
3	IE_ALMOSTFULL	Enables an interrupt if the fill level of the FIFO is greater than the value of the almostfull register.
2	IE_ALMOSTEMPTY	Enables an interrupt if the fill level of the FIFO is less than the value of the almostempty register.
4	IE_OVERFLOW	Enables an interrupt if the FIFO overflows. The FIFO overflows when an Avalon write master writes to a full FIFO.

Table 5–8. InterruptEnable Bit Field Descriptions (Part 2 of 2)

Bit(s)	Name	Description
5	IE_UNDERFLOW	Enables an interrupt is the FIFO underflows. The FIFO underflows when an Avalon read master reads from an empty FIFO.
6	ALL	Enables all 6 status conditions to interrupt.

Macros to access all of the registers are defined in **altera_avalon_fifo_regs.h.** For example, this file includes the following macros to access the `status` register.

```
#define ALTERA_AVALON_FIFO_LEVEL_REG          0
#define ALTERA_AVALON_FIFO_STATUS_REG         1
#define ALTERA_AVALON_FIFO_EVENT_REG          2
#define ALTERA_AVALON_FIFO_IENABLE_REG        3
#define ALTERA_AVALON_FIFO_ALMOSTFULL_REG     4
#define ALTERA_AVALON_FIFO_ALMOSTEMPTY_REG    5
```



For a complete list of predefined macros and utilities to access the on-chip FIFO hardware, see: **<*install_dir*>\quartus\sopc_builder\components\altera_avalon_fifo\HAL\inc\alatera_avalon_fifo.h** and **<*install_dir*>\quartus\sopc_builder\components\altera_avalon_fifo\HAL\inc\alatera_avalon_fifo_util.h.**

Software Example

An extensive programming example for the on-chip FIFO memory appears next to this document on the Quartus II literature page. Visit **www.altera.com/literature/quartus2/lit-qts-peripherals.jsp**.

Example 5–1. Sample Code for the On-Chip FIFO Memory

```
/************************************************************************/
//Includes
#include "altera_avalon_fifo_regs.h"
#include "altera_avalon_fifo_util.h"
#include "system.h"
#include "sys/alt_irq.h"
#include <stdio.h>
#include <stdlib.h>

#define ALMOST_EMPTY 2
#define ALMOST_FULL OUTPUT_FIFO_OUT_FIFO_DEPTH-5

volatile int input_fifo_wrclk_irq_event;

void print_status(alt_u32 control_base_address)
{
    printf("---------------------------------------\n");
    printf("LEVEL = %u\n",
altera_avalon_fifo_read_level(control_base_address) );
    printf("STATUS = %u\n",
altera_avalon_fifo_read_status(control_base_address,
ALTERA_AVALON_FIFO_STATUS_ALL) );
    printf("EVENT = %u\n",
altera_avalon_fifo_read_event(control_base_address,
ALTERA_AVALON_FIFO_EVENT_ALL) );
    printf("IENABLE = %u\n",
altera_avalon_fifo_read_ienable(control_base_address,
ALTERA_AVALON_FIFO_IENABLE_ALL) );
    printf("ALMOSTEMPTY = %u\n",
altera_avalon_fifo_read_almostempty(control_base_address) );
    printf("ALMOSTFULL = %u\n\n",
altera_avalon_fifo_read_almostfull(control_base_address));
}

static void handle_input_fifo_wrclk_interrupts(void* context, alt_u32 id)
{
    /* Cast context to input_fifo_wrclk_irq_event's type. It is important
     * to declare this volatile to avoid unwanted compiler optimization.
     */
   volatile int* input_fifo_wrclk_irq_event_ptr = (volatile int*) context;

    /* Store the value in the FIFO's irq history register in *context. */
    *input_fifo_wrclk_irq_event_ptr =
altera_avalon_fifo_read_event(INPUT_FIFO_IN_CSR_BASE,
ALTERA_AVALON_FIFO_EVENT_ALL);
    printf("Interrupt Occurs for %#x\n", INPUT_FIFO_IN_CSR_BASE);
    print_status(INPUT_FIFO_IN_CSR_BASE);

    /* Reset the FIFO's IRQ History register. */
    altera_avalon_fifo_clear_event(INPUT_FIFO_IN_CSR_BASE,
```

```
ALTERA_AVALON_FIFO_EVENT_ALL);
}

/* Initialize the fifo */
static int init_input_fifo_wrclk_control()
{
    int return_code = ALTERA_AVALON_FIFO_OK;

    /* Recast the IRQ History pointer to match the alt_irq_register()
function
     * prototype. */
    void* input_fifo_wrclk_irq_event_ptr = (void*)
&input_fifo_wrclk_irq_event;
    /* Enable all interrupts. */

    /* Clear event register, set enable all irq, set almostempty and
almostfull threshold */
    return_code = altera_avalon_fifo_init(INPUT_FIFO_IN_CSR_BASE,
                                          0, // Disabled interrupts
                                          ALMOST_EMPTY,
                                          ALMOST_FULL);

    /* Register the interrupt handler. */
    alt_irq_register( INPUT_FIFO_IN_CSR_IRQ,
input_fifo_wrclk_irq_event_ptr, handle_input_fifo_wrclk_interrupts );
    return return_code;
}
```

On-Chip FIFO Memory API

This section describes the application programming interface (API) for the on-chip FIFO memory core.

altera_avalon_fifo_init()

Prototype:	`int altera_avalon_fifo_init(alt_u32 address, alt_u32 ienable, alt_u32 emptymark, alt_u32 fullmark)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `ienable`—the value to write to the `interruptenable` register `emptymark`—the value for the almost empty threshold level `fullmark`—the value for the almost full threshold level
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_EVENT_CLEAR_ERROR` for clear errors, `ALTERA_AVALON_FIFO_IENABLE_WRITE_ERROR` for interrupt enable write errors, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` for errors writing the `almostfull` and `almostempty` registers.
Description:	Clears the `event` register, writes the `interruptenable` register, and sets the `almostfull` register and `almostempy` registers.

altera_avalon_fifo_read_status()

Prototype:	`int altera_avalon_fifo_read_status(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `mask`—masks the read value from the status register
Returns:	Returns the fill level of the FIFO.
Description:	Gets the fill level of the FIFO which is the AND of the value of the addressed register and the mask.

altera_avalon_fifo_read_ienable()

Prototype:	`int altera_avalon_fifo_read_ienable(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `mask`—masks the read value from the `interruptenable` register
Returns:	Returns the logical `AND` of the `interruptenable` register and the mask.
Description:	Gets the logical `AND` of the `interruptenable` register and the mask.

altera_avalon_fifo_read_almostfull()

Prototype:	`int altera_avalon_fifo_read_almostfull(alt_u32 address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave
Returns:	Returns the value of the `almostfull` register.
Description:	Gets the value of the `almostfull` register.

altera_avalon_fifo_read_almostempty()

Prototype:	`int altera_avalon_fifo_read_almostempty(alt_u32 address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave
Returns:	Returns the value of the `almostempty` register.
Description:	Gets the value of the `almostempty` register.

altera_avalon_fifo_read_event()

Prototype:	`int altera_avalon_fifo_read_event(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `mask`—masks the read value from the event register
Returns:	Returns the logical `AND` of the `event` register and the mask.
Description:	Gets the logical `AND` of the `event` register and the mask. To read single bits of the event register use the single bit masks, for example: ALTERA_AVALON_FIFO_FIFO_EVENT_F_MSK. To read the entire event register use the full mask: ALTERA_AVALON_FIFO_EVENT_ALL.

altera_avalon_fifo_read_level()

Prototype:	`int altera_avalon_fifo_read_level(alt_u32 address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave
Returns:	Returns the fill level of the FIFO.
Description:	Gets the fill level of the FIFO.

altera_avalon_fifo_clear_event()

Prototype:	`int altera_avalon_fifo_clear_event(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `mask`—the mask to use for bit-clearing (1 means clear this bit, 0 means don't)
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_EVENT_CLEAR_ERROR` if unsuccessful.
Description:	Clears the specified bits of the `event` register.

altera_avalon_fifo_write_ienable()

Prototype:	`int altera_avalon_fifo_write_ienable(alt_u32 address, alt_u32 mask`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `mask`—the value to write to the `interruptenable` register. See **altera_avalon_fifo_regs.h** for individual interrupt bit masks.
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_IENABLE_WRITE_ERROR` if unsuccessful.
Description:	Writes the specified bits of the `interruptenable` register.

altera_avalon_fifo_write_almostfull()

Prototype:	`int altera_avalon_fifo_write_almostfull(alt_u32 address, alt_u32 data)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `data`—the value for the almost full threshold level
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` if unsuccessful.
Description:	Writes `data` to the `almostfull` register.

altera_avalon_fifo_write_almostempty()

Prototype:	`int altera_avalon_fifo_write_almostempty(alt_u32 address, alt_u23 data)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `data`—the value for the almost empty threshold level
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` if unsuccessful.
Description:	Writes `data` to the `almostempty` register.

altera_avalon_write_fifo()

Prototype:	`int altera_avalon_write_fifo(alt_u32 write_address, alt_u32 ctrl_address, alt_u32 data)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`write_address`—the base address of the FIFO write slave `ctrl_address`—the base address of the FIFO control slave `data`—the value to write to address offset 0 for Avalon-MM to Avalon-ST transfers, the value to write to the single address available for Avalon-MM t o Avalon-MM transfers. See the *Avalon Memory-Mapped and Avalon Streaming Interface Specifications* for the data ordering.
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_FULL` if unsuccessful.
Description:	Writes `data` to the specified address if the FIFO is not full.

altera_avalon_write_other_info()

Prototype:	`int altera_avalon_write_other_info(alt_u32 write_address, alt_u32 ctrl_address, alt_u32 data)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`write_address`—the base address of the FIFO write slave `ctrl_address`—the base address of the FIFO control slave `data`—the packet status information to write to address offset 1 of the Avalon interface. See the *Avalon Memory-Mapped and Avalon Streaming Interface Specifications* for the ordering of the packet status information.
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_FULL` if unsuccessful.
Description:	Writes the packet status information to the `write_address`. Only valid when **Enable packet data** is on.

altera_avalon_fifo_read_fifo()

Prototype:	`int altera_avalon_fifo_read_fifo(alt_u32 read_address, alt_u32 ctrl_address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`read_address`—the base address of the FIFO read slave `ctrl_address`—the base address of the FIFO control slave
Returns:	Returns the data from address offset 0, or 0 if the FIFO is empty.
Description:	Gets the data addressed by read_address.

altera_avalon_fifo_read_other_info()

Prototype:	`int altera_avalon_fifo_read_other_info(alt_u32 read_address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`read_address`—the base address of the FIFO read slave
Returns:	Retunrs the packet status information from address offset 1 of the Avalon interface. See the *Avalon Memory-Mapped and Avalon Streaming Interface Specifications* for the ordering of the packet status information.
Description:	Reads the packet status information from the specified `read_address`. Only valid when **Enable packet data** is on.

Referenced Documents

This chapter references the Avalon Streaming Interface Specification.

Document Revision History

Table 5–9 shows the revision history for this chapter.

Table 5–9. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 5 was formerly Chapter 4.	—
May 2007 v7.1.0	Initial release.	—

6. Scatter-Gather DMA Controller Core

QII55003-7.2.0

Core Overview

The Scatter-Gather direct memory access (SG-DMA) controller core implements high-speed data transfer between two devices. The SG-DMA core can be used to transfer data from the following sources:

- Memory to memory
- Data stream to memory
- Memory to data stream

The SG-DMA controller core transfers and merges non-contiguous memory to a continuous address space. It also transfers contiguous memory to non-contiguous memory. The core operates by reading a series of descriptors that specify the data to be transferred.

For applications requiring more than one DMA channel, multiple instantiations of the core can provide the required throughput. Each SG-DMA controller has its own series of descriptors specifying the data transfers. A single software module controls all of the DMA channels.

The SG-DMA controller core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. For the Nios® II processor, device drivers are provided in the HAL system library, allowing software to access the core using the ANSI C Standard Library **stdio.h** routines.

Example Systems

Figure 6–1 shows a SG-DMA controller core in a block diagram for the DMA subsystem of a printed circuit board. The SG-DMA core in the FPGA reads streaming data from an internal streaming component and writes data to an external memory. A Nios II processor provides overall system control. The descriptor table, containing the linked list of descriptors specifying data transfers to be executed, can be located in the FPGA or an external memory. Locating this table in an external memory will free up resources in the FPGA; however, an external descriptor table will increase the overhead involved when the descriptor processor reads and updates the table. The SG-DMA core has an internal FIFO to store descriptors read from memory, which allows it to perform descriptor read, execute, and write back operations in parallel, hiding the descriptor read latency.

Figure 6–1. Scatter-Gather DMA Controller Core with Streaming Peripheral and External Memory

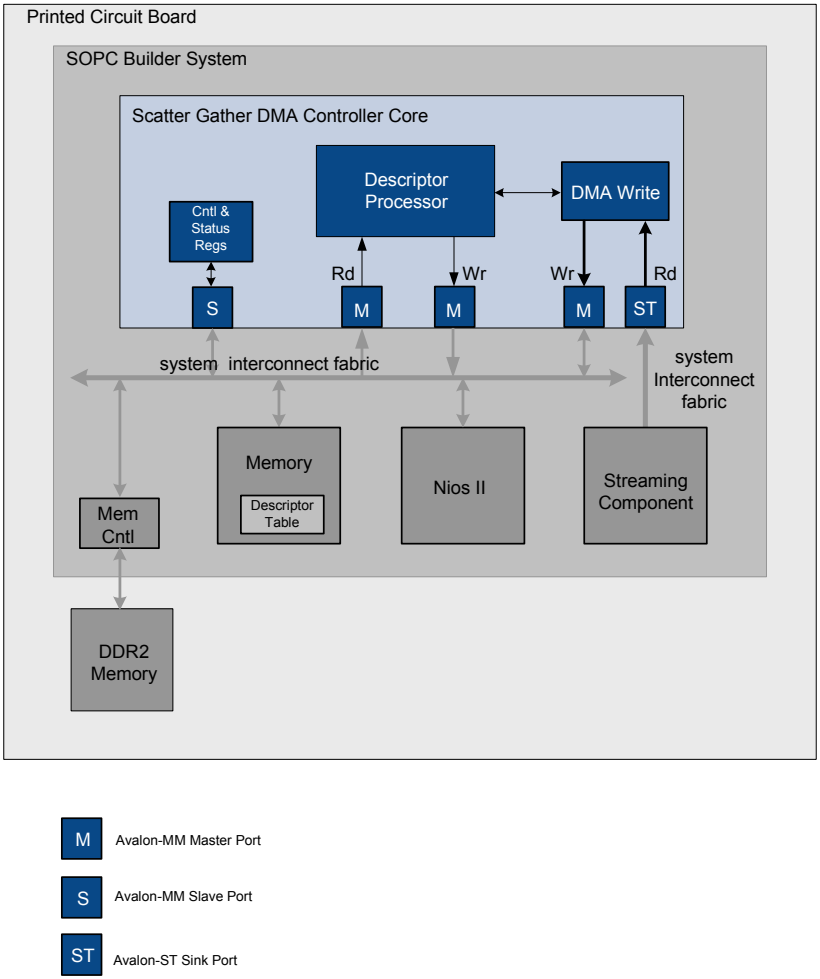
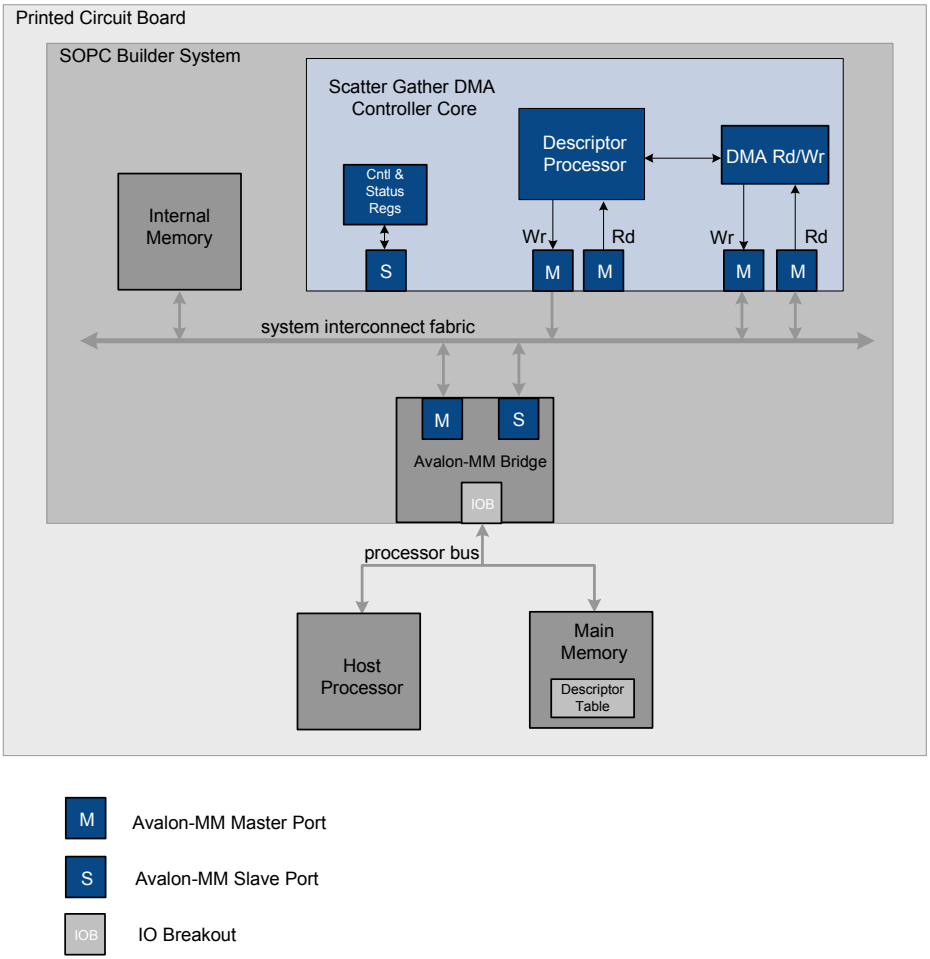


Figure 6–2 shows a different use of the SG-DMA controller core. In Figure 6–2, SG-DMA core transfers data between an internal and external memory. The host processor and memory are on the system bus, typically either a PCI-Express or Serial RapidIO™.

Figure 6–2. Scatter-Gather DMA Controller Core with Internal and External Memory



Figures 6–1 and 6–2 illustrate systems using the SG-DMA controller core and omit some of the internals of the core itself. Figure 6–3 on page 6–7 illustrates all of the SG-DMA controller core internals.

Resource Usage and Performance

Resource utilization for the core is 600–1400 LEs, depending upon the width of the datapath, the parameterization of the core, and the type of data transfer. Table 6–1 provides resource utilization for a SG-DMA

controller core used for memory to memory transfer. The core is highly parameterized and the resource utilization will vary with the configuration specified.

Table 6–1. SG-DMA Estimated Resource Usage

Datapath	Cyclone® II	Stratix® (Approx. LEs)	Stratix II (Approx. ALUTs)
8-bit datapath	850	650	600
32-bit datapath	1100	850	700
64-bit datapath	1250	1250	800

The core operating frequency varies with the device and the size of the datapath. Table 6–2 provides an example of expected performance for SG-DMA cores instantiated in several different device families.

Table 6–2. SG-DMA Performance Estimates

Device	Datapath	f_{MAX}	Throughput
Cyclone II	64 bits	150 MHz	9.6 Gbps
Cyclone III	64 bits	160 MHz	10.2 Gbps
Stratix II/Stratix II GX	64 bits	250 MHz	16.0 Gbps
Stratix III	64 bits	300 MHz	19.2 Gbps

Comparison of SG-DMA Controller Core and DMA Controller Core

The SG-DMA controller core provides a significant performance enhancement over the previously available DMA controller core, which could only perform a single DMA transfer at a time. With the older DMA controller core, a CPU performed separate reads for each entry of the DMA descriptor table and then executed separate IO writes to program the DMA controller to perform the transfer. Transfers to non-contiguous memory could not be linked; consequently, the CPU overhead was substantial for small transfers, degrading overall system performance. In contrast, the SG-DMA controller core reads a series of descriptors from memory that describe the required transactions and performs all of the transfers without additional intervention from the CPU.

Functional Description

The SG-DMA controller core comprises three major blocks: a descriptor processor, a DMA read block and a DMA write block. (See Figure 6–3 on page 6–7.) These blocks are combined to create three different configurations:

- memory to memory
- memory to stream
- stream to memory

For the memory-to-memory configuration, the core includes all three blocks. If the core is configured for memory-to-stream transactions, only the descriptor processor and read blocks are required. If the core is configured for stream-to-memory transactions, only the descriptor processor and write blocks are required. In the memory-to-memory configuration, an internal FIFO holds data being transferred between the read and write blocks. In the other two configurations, an external FIFO might be required depending upon the throughput of the components being connected. For designs requiring an external FIFO, the on-chip FIFO memory available in SOPC Builder can be used.

The following sections describe the three configurations of the SG-DMA controller core and the behavior of the internal modules for each configuration.

Memory-to-Memory Configuration

As Figure 6–3 illustrates, the memory-to-memory configuration includes five Avalon-MM ports. The descriptor processor block uses read and write Avalon-MM master ports to access and update descriptors. The DMA read block has an Avalon-MM master read port to read data from memory; it has an Avalon-ST port to pass data to the DMA write block. The DMA write block has an Avalon-ST port to receive data from the read block and an Avalon-MM master write port to write the data to memory. Software accesses an Avalon-MM slave port to read and write the `control` and `status` registers.

In the memory-to-memory configuration, the descriptor processor reads descriptors from the descriptor table and pushes the appropriate commands onto the input FIFOs of the DMA read and write blocks. It also receives a status token from the read or write block after each descriptor has been processed. The status token contains information about the status of the transfer, including the number of bytes transferred. The descriptor processor then writes this information back to the appropriate entry in the descriptor table.

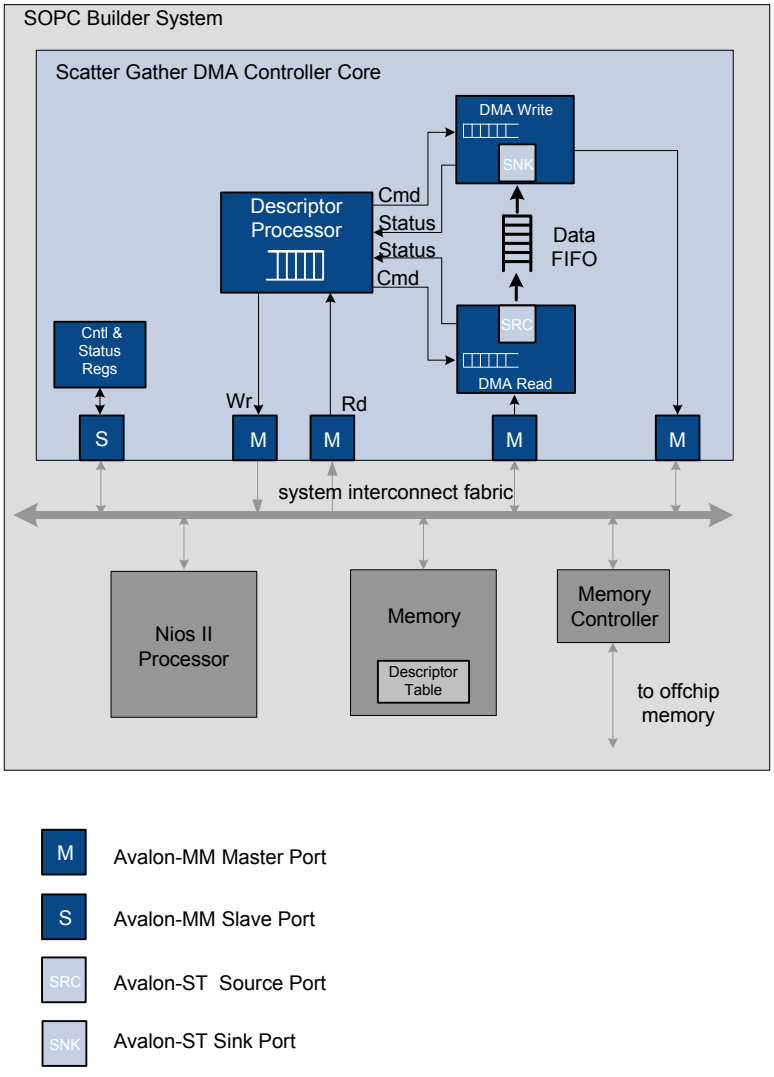
In the memory-to-memory configuration, an internal data FIFO stores data being transferred between the read and write blocks to provide buffering and flow control.

To execute a DMA read transfer between memories, the following steps are executed:

1. Software writes the descriptor into memory.
2. Software writes the location of the first descriptor address to SG-DMA controller hardware and initiates the transfer by setting the RUN bit of the SG-DMA `control` register.
3. The descriptor processor reads the descriptors from memory and writes them into a command FIFO which feeds commands to both the DMA read and write blocks.
4. The DMA write block gets the destination address from its command FIFO. The write block continues to execute writes until the specified number of bytes have been transferred. It then sends a status update to the DMA controller. If the data FIFO ever empties, the write block pauses until the FIFO has more data to write.
5. The descriptor processor updates the appropriate entry in the descriptor table.

Figure 6–3 illustrates one possible configuration for the memory-to-memory SG-DMA controller with an internal Nios II processor and descriptor table.

Figure 6–3. Scatter-Gather DMA Controller Core for Memory-to-Memory Configuration



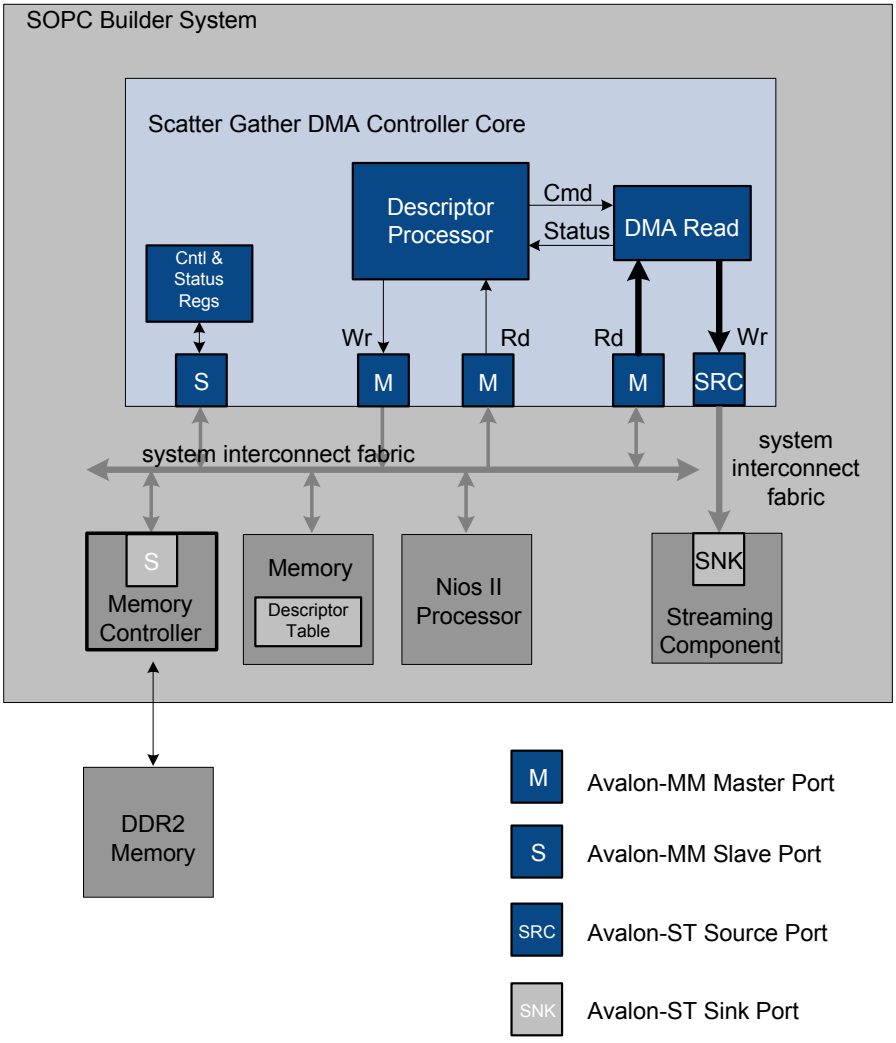
Memory-to-Stream Configuration

The memory-to-stream configuration includes the descriptor processor and the DMA read block. As Figure 6–3 illustrates, this configuration includes four Avalon-MM ports and one Avalon-ST port. The descriptor processor block includes read and write Avalon-MM master ports to

access and update descriptors. The DMA read block has an Avalon-MM master read port to read data from memory and an Avalon-ST port to write the data to a streaming component. An Avalon-MM slave port is used to read and write the `control` and `status` registers.

Figure 6–4 on page 6–9 illustrates a SG-DMA controller in the memory-to-stream configuration. In this example, the Nios II processor and descriptor table are inside the FPGA. Data from an external DDR2 memory is read by the SG-DMA controller and written to an internal streaming peripheral. The read block returns status to the descriptor processor upon completion of each descriptor.

Figure 6–4. Scatter-Gather DMA Controller Memory-to-Stream Configuration



The transfer operation includes the following steps:

1. Software writes the descriptor into memory.
2. Software writes the location of first descriptor address to SG-DMA controller hardware and initiates the transfer by setting the RUN bit of the SG-DMA `control` register.

3. The descriptor processor reads the descriptors from memory and writes them into the input command FIFO in the read block.

4. The read block reads from the source address and pushes the data to its stream port. The read block continues reads until the specified number of bytes have been transferred. It then sends a status update to the descriptor processor.

5. The descriptor processor updates the appropriate entry in the descriptor table.

Stream-to-Memory Configuration

The stream-to-memory configuration includes the descriptor processor and the DMA write block. The write block returns status to the descriptor processor upon completion of each descriptor. This configuration is similar to the memory-to-stream configuration except that the data flows from a streaming component to a memory device as Figure 6–5 illustrates. In this example, an On-Chip FIFO Memory component is used to provide a buffer between the streaming component and the DMA write.

The transfer operation includes the following steps:

1. Software writes the descriptor into memory.

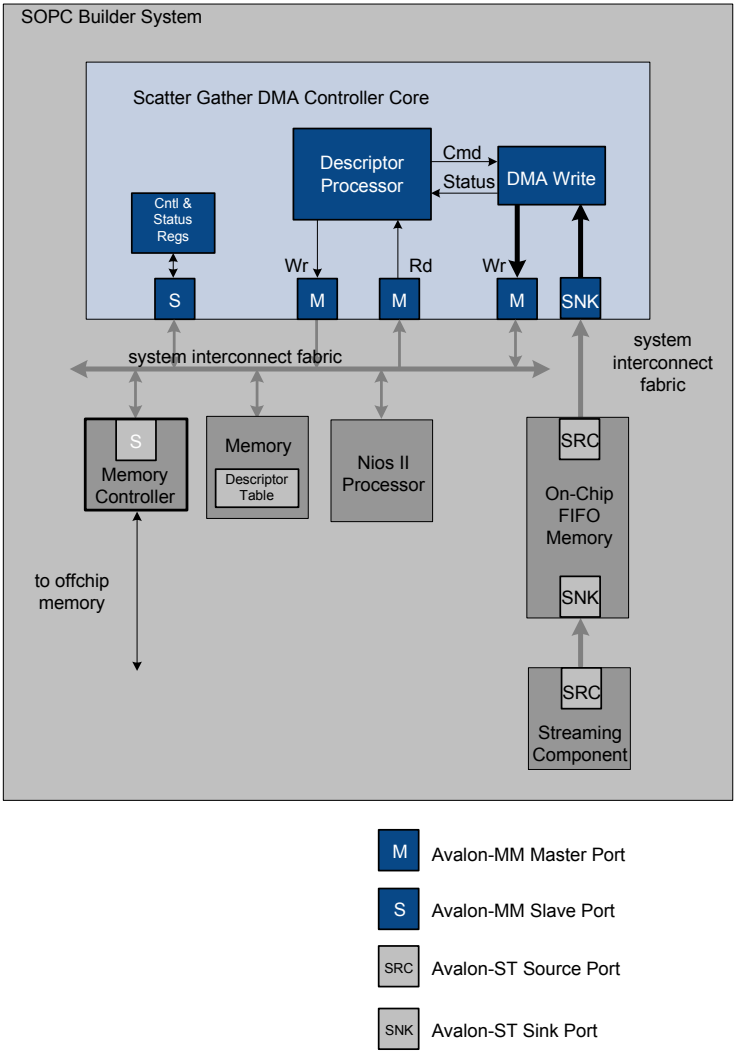
2. Software writes the location of the first descriptor address to SG-DMA controller hardware and initiates the transfer by writing the RUN bit of the SG-DMA `control` register.

3. The descriptor processor reads the descriptors from memory and writes them into the write block.

4. The write block reads from its stream port and writes the data to its Avalon master port. The write block continues reads until the specified number of bytes have been transferred. It then sends a status update to the descriptor processor.

5. The descriptor processor updates the appropriate entry in the descriptor table.

Figure 6–5. Scatter-Gather DMA Controller Stream-to-Memory Configuration



Possible Sources of Errors

The SG-DMA core has a parameterizable error width. Error signals are wired directly to the Avalon-ST source or sink to which the SG-DMA core is connected. Table 6–3 lists the error signals when the core is operating in the memory to stream configuration and connected to the transmit FIFO interface of the Altera Triple-Speed Ethernet MegaCore®.

Table 6–3. Avalon-ST Transmit Channel Error Types

Signal Type	Description
TSE_transmit_error[0]	Transmit Frame Error. Asserted to indicate that the transmitted frame should be viewed as invalid by the Ethernet MAC. The frame is then transferred onto the GMII interface with an error code during the frame transfer.

Table 6–4 lists the error signals when the core is operating in the stream-to-memory configuration and connected to the transmit FIFO interface of the Altera Triple-Speed Ethernet MegaCore.

Table 6–4. Avalon-ST Receive Channel Error Types

Signal Type	Description
TSE_receive_error[0]	Receive Frame Error. This signal indicates that an error has occurred. It is the logical OR of receive errors 1 through 5.
TSE_receive_error[1]	Invalid Length Error. Asserted when the received frame has an invalid length as defined by the IEEE 802.3 standard.
TSE_receive_error[2]	CRC Error. Asserted when the frame has been received with a CRC-32 error.
TSE_receive_error[3]	Receive Frame Truncated. Asserted when the received frame has been truncated due to receive FIFO overflow.
TSE_receive_error[4]	Received Frame corrupted due to PHY error. (The PHY has asserted an error on the receive GMII interface.)
TSE_receive_error[5]	Collision Error. Asserted when the frame was received with a collision.

Detailed Description of Each Block

The following sections provide a detailed description of each functional block.

Descriptor Processor Block

The descriptor processor reads descriptors from memory using its Avalon-MM descriptor read master port and pushes commands onto the command FIFOs of the DMA read and write blocks. The command includes the following fields to specify the transfers:

- source address
- destination address
- read size
- write size
- bytes to transfer
- increment read address after each transfer
- increment write address after each transfer
- generate end of packet

DMA Read Block

The DMA read block reads commands from its input command FIFO. For each command, it reads data from the source address on its Avalon-MM port. In the memory-to-memory configuration, it pushes the data into the data FIFO. In the memory-to-stream configuration, it immediately writes the data to the Avalon-ST source port.



The DMA read block will not begin an Avalon-MM read unless its data FIFO has enough space to store all of the data read. This restriction requires the external FIFO be at least as deep as the maximum supported read size.

DMA Write Block

The DMA write block reads commands from its input command FIFO. For each command, it writes data received on its Avalon-ST sink port to the destination address. In the memory-to-memory and the stream-to-memory configurations, it reads the data from its Avalon-ST port and writes to its Avalon-MM port.

Device Support and Tools

The SG-DMA controller core supports the Arria™ GX, Stratix III, Stratix II GX, Stratix II, Stratix, Cyclone III, Cyclone II, Cyclone and Hardcopy® II device families.

Instantiating the Core in SOPC Builder

Hardware designers use the MegaWizard interface for the SG-DMA controller core in SOPC Builder to specify the core configuration.



If an SOPC Builder system contains both the SG-DMA controller and JTAG UART cores, set the IRQs for the SG-DMA controller core to a higher priority than the IRQs for the JTAG UART core.

The following sections describe the available options in the configuration wizard.

Transfer Mode

This list allows you to select between the **Memory To Memory**, **Memory To Stream**, and **Stream To Memory** configurations. For more information about these configurations, see "Memory-to-Memory Configuration" on page 6–5, "Memory-to-Stream Configuration" on page 6–7, and "Stream-to-Memory Configuration" on page 6–10.

Allow Unaligned Transfers

If **Allow unaligned transfers** is on, data transfers for data widths that are not a power of two will be aligned on word boundaries. Unaligned transfers require extra logic that may negatively impact system performance.

Data and Error Widths

The **Data width** list allows you to select data width in bits for the Avalon-MM read and write ports. The **Source error width** and **Sink error width** lists allow you to select widths of the error signals for the Avalon-ST source and sink ports.

FIFO Depth

The **Data transfer FIFO depth** list sets the depth for all three descriptor FIFOs:

- the Descriptor Processor block FIFO
- the DMA read block FIFO
- the DMA write block FIFO

The **Data transfer FIFO depth** list also sets the depth for the internal data FIFO used in the memory-to-memory configuration. These FIFOs are all illustrated in Figure 6–3 on page 6–7.

Hardware Simulation Considerations

Signals for hardware simulation are automatically generated and show up as part of the Nios II simulation accessible from the Nios II IDE. On the Run menu, point to Run As and click **Nios II Modelsim**.

Software Programming Model

The following sections describe the software programming model for the SG-DMA controller core.

HAL System Library Support

The Altera-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access the SG-DMA controller core via the familiar HAL API and the ANSI C standard library.

Software Files

The SG-DMA controller provides the following software files. These files provide low-level access to the hardware and drivers that integrate into the Nios II HAL system library. Application developers should not modify these files.

- **altera_avalon_sgdma_regs.h**—defines the core's register map, providing symbolic constants to access the low-level hardware
- **altera_avalon_sgdma.h**—provides definitions for the Altera Avalon SG-DMA buffer control and status flags.
- **altera_avalon_sgdma.c**—provides function definitions for the code that implements the SG-DMA controller core.
- **altera_avalon_sgdma_descriptor.h**—defines the core's descriptor, providing symbolic constants to access the low-level hardware.

Programming with the SG-DMA Controller

This section describes the software constructs for programming the SG-DMA Controller.

Table 6–5. SG-DMA Controller Functions

Function Name	Description
`alt_avalon_sgdma_do_async_transfer()`	Sets up and begins a non-blocking transfer of one or more descriptors or a descriptor chain.
`alt_avalon_sgdma_do_sync_transfer()`	Sends a fully formed descriptor, or list of descriptors, to the SG-DMA Controller for transfer. This function will block both before transfer if the controller is busy and until the requested transfer has completed.
`alt_avalon_sgdma_construct_mem_to_mem_desc()`	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-MM transfer.
`alt_avalon_sgdma_construct_stream_to_mem_desc()`	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-ST to Avalon-MM transfer.
`alt_avalon_sgdma_construct_mem_to_stream_desc()`	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-ST transfer.
`alt_avalon_sgdma_check_descriptor_status()`	Reads the `status` register of the descriptor.
`alt_avalon_sgdma_register_callback()`	Associates a user-specific routine with the SG-DMA interrupt handler.
`alt_avalon_sgdma_start()`	Starts the DMA engine.
`alt_avalon_sgdma_stop()`	Stops the DMA engine.
`alt_avalon_sgdma_open()`	Retrieves a pointer to the SG-DMA controller with the given name.

Software Control

The host processor programs the SG-DMA by writing to its `control` register. The host processor reads SG-DMA `status` register to determine the current status.

Table 6–6 shows the offsets for the control and status registers.

Table 6–6. SG-DMA Control and Status Slave Register Map

Offset	Register Name
base + 0	status
base + 1	control
base + 2	next_descriptor_pointer

Software writes the control register to specify the behavior of the SG-DMA controller. This register determines the conditions under which the SG-DMA controller will generate an interrupt. It also includes the control bits used to start and stop processing descriptors.



Bursting to the control port is not supported.

Table 6–7 provides a bit-map for the control register.

Table 6–7. SG-DMA Control Register Map (Part 1 of 2)

Bit	Bit Name	Rd/Wr/Clr	Description
0	IE_ERROR	R/W	Assert an interrupt when (ERROR = 1).
1	IE_EOP_ENCOUNTERED	R/W	Assert an interrupt when (EOP_ENCOUNTERED = 1).
2	IE_DESCRIPTOR_COMPLETED	R/W	Assert an interrupt when (DESCRIPTOR_COMPLETED = 1).
3	IE_CHAIN_COMPLETED	R/W	Assert an interrupt when (CHAIN COMPLETED =1).
4	IE_GLOBAL	R/W	Global signal to enable all interrupts.
5	RUN	R/W	The SG-DMA processes descriptors in its queue as long as RUN = 1. The SG-DMA will not process the next descriptor in its queue when RUN = 0. Setting RUN starts the descriptor processor that initiates DMA transactions. Clearing RUN will not stop processing of a descriptor if processing has already begun.
6	STOP_DMA_ER	R/W	Stops DMA after current descriptor if ERROR is detected.
7	IE_MAX_DESC_ PROCESSED *(1)*	R/W	Enables interrupts when MAX_DESC_PROCESSED is reached.
8 .. 15	MAX_DESC_ PROCESSED *(1)*	R/W	Specifies the number of descriptors to process before invoking interrupt.

Table 6–7. SG-DMA Control Register Map (Part 2 of 2)

Bit	Bit Name	Rd/Wr/Clr	Description
16	SW_RESET	R/W	Resets the SG-DMA hardware and stops all operations immediately.
17	PARK	R/W	Enables the hardware to repeatedly use the same descriptor without software intervention. The bit owned_by_hw is not cleared, thus allowing the hardware to reuse the descriptor.
18..30	Reserved		
31	CLEAR INTERRUPT	R/W	Set this bit to 1 to clear pending interrupts.

Note to Table 6–7:
(1) Available if interrupt coalescing is selected in the synthesis options.

Table 6–8 provides a bit-map for the status register.

Table 6–8. SG-DMA Status Register Map

Bit	Bit Name	Rd/Wr/Clr	Description
0	ERROR	R/C *(1)(2)*	Avalon-ST error encountered during transfer.
1	EOP_ENCOUNTERED	R/C	Transfer terminated by Avalon-ST EOP.
2	DESCRIPTOR_COMPLETED	R/C *(1)(2)*	A descriptor was processed to completion.
3	CHAIN_COMPLETED	R/C *(1)(2)*	Unable to process next descriptor because Owned by HW = 0.
4	BUSY	R/C *(1)(3)*)	Indicates that descriptors are being processed; the linked list of descriptors is not yet completed.
5 .. 31	reserved		

Notes to Table 6–8:
(1) This bit must be cleared after a read is performed. Write one to clear this bit.
(2) This bit is updated by hardware after each DMA transfer completes. It remains set until software writes one to clear.
(3) This bit is continuously updated by the hardware.

Next Descriptor Pointer

Software writes the address of the first descriptor to this register as part of the system initialization sequence. When RUN = 1, the SG-DMA updates this register with the location of the next descriptor to be fetched. To stop execution of the SG-DMA core, software clears the RUN bit. When RUN is 0, the SG-DMA hardware completes the data transfers for the current descriptor and then stops processing. Software can then modify the remaining linked-list and restart the SG-DMA hardware.



While BUSY = 1, the next descriptor pointer is updated by hardware. The next descriptor pointer can only be reliably read by software when BUSY = 0.

DMA Descriptors

The DMA descriptors specify all information required to perform data transfers, including: the source address, destination address, and the number of bytes to be transferred. The descriptors are stored in a table that is written by software. This table can be stored in FPGA memory or an external memory device as a linked list. The descriptor must be initialized by user applications and aligned on a 256-bit boundary.

Table 6–9 shows the layout of a descriptor entry.

Table 6–9. Descriptor Layout

Offset	Bit Field Names			
	31 24	23 16	15 8	7 0
base	SOURCE			
base + 1	RESERVED			
base + 2	DESTINATION			
base + 3	RESERVED			
base + 4	NEXT_DESC_PTR			
base + 5	RESERVED			
base + 6	WRITE_	READ_	BYTES_TO_TRANSFER	
base + 7	DESC_CONTROL	DESC_STATUS	ACTUAL_BYTES_TRANSFERRED	

Table 6–10 describes the function of the various fields.

Table 6–10. Descriptor Field Descriptions (Part 1 of 2)

Field Name	Rd/Wr/Clr	Description
SOURCE	R/W	Specifies the address of data to be read. This address is set to 0 if the input source is an Avalon Streaming (Avalon-ST) interface.
DESTINATION	R/W	Specifies the address to which data should be written. This address is set to 0 if the write data is an Avalon-ST interface.
NEXT_DESC_PTR	R/W	Specifies the next descriptor in the linked list.

Table 6–10. Descriptor Field Descriptions (Part 2 of 2)

Field Name	Rd/Wr/Clr	Description
BYTES_TO_TRANSFER	R/W	Specifies the number of bytes to transfer. If BYTES_TO_TRANSFER = 0, the transaction will be terminated by an EOP.
READ_	R/W	
WRITE_	R/W	
ACTUAL_BYTES_TRANSFERRED	R	Specifies the number of bytes that are successfully transferred by the DMA hardware.
DESC_CONTROL	R/W	See Table 6–12 for descriptions of each bit.
DESC_STATUS	R/W	See Table 6–11 for descriptions of each bit.

The descriptor processor reads the DESC_CONTROL fields to determine how to proceed with the DMA transaction. Table 6–11 provides a bit-map for theses fields.

Table 6–11. Desc_Control Field Map

Bits	Field Name	Rd/Wr/Clr	Description
0	Generate_EOP	W	When set, DMA Read should generate an EOP on the final word.
1	Read_Fixed_Address	R/W	For Avalon-MM ports, when set to 1, DMA Read does not increment the memory address. When 0, the read address increments after each read. When used in Memory-to-Stream mode, the read engine generates a startofpacket signal on the first word.
2	Write_Fixed_Address	R/W	Used only for Avalon-MM ports. When set to 1, DMA Write does not increment the memory address. When 0, the write address increments after each write.
3 .. 6	Avalon-ST_Channel_Number	R/W	DMA Read drives this value onto the Avalon-ST channel port for each word in the transaction. The DMA Write replaces this value with the Avalon-ST channel number for its sink port.
7	Owned_by_HW	R/W	This bit determines whether hardware or software has write access to the descriptor of the SG-DMA control and status register. When Owned_by_HW=1 the hardware can update this pointer. When Owned_by_HW=0, software can update this pointer.

After completing a DMA transaction, the descriptor processor updates the DESC_STATUS fields to indicate how the transaction proceeded. The error conditions these fields record can only occur on an Avalon-ST interface. Table 6–12 provides a bit-map for the DESC_STATUS fields.

Table 6–12. Descriptor Desc_Status Bit Map

Bit	Bit Name	Rd/Wr/Clr	Description
0	E_CRC	R	When set, indicates that a CRC error occurred on the Avalon-ST interface.
1	E_PARITY	R	When set, indicates that a parity error occurred on the Avalon-ST interface.
2	E_OVERFLOW	R	When set, indicates that an overflow occurred on the Avalon-ST interface.
3	E_SYNC	R	When set, indicates that an out-of-sync error occurred on the Avalon-ST interface.
4	E_UEOP	R	When set, indicates that an unexpected EOP error occurred on the Avalon-ST interface.
5	E_MEOP	R	When set, indicates that a missing EOP error occurred on the Avalon-ST interface.
6	E_MSOP	R	When set, indicates that a missing SOP error occurred on the Avalon-ST interface.
7	Terminated_by_EOP	R	When set, indicates that a write transaction was terminated by EOP.

Macros to access all of the registers are defined in **altera_avalon_sgdma_regs.h.** For example, this file includes macros to access the status register, including the following macros:

```
#define IOADDR_ ALTERA_AVALON_SGDMA_STATUS(base)        __IO_CALC_ADDRESS_DYNAMIC(base, 0)
#define IORD_ALTERA_AVALON_SGDMA_STATUS(base)          IORD(base, 0)
#define IOWR_ALTERA_AVALON_SGDMA_STATUS(base, data)    IOWR(base, 0, data)
#define ALTERA_AVALON_SGDMA_STATUS_ERROR_MSK (0x1)
#define ALTERA_AVALON_SGDMA_STATUS_ERROR_OFST (0)
#define ALTERA_AVALON_SGDMA_STATUS_EOP_ENCOUNTERED_MSK (0x2)
#define ALTERA_AVALON_SGDMA_STATUS_EOP_ENCOUNTERED_OFST (1)
```

For a complete list of predefined macros and utilities to access the SG-DMA Controller hardware, see:

- **<*install_dir*>\quartus\sopc_builder\components\altera_avalon_sgdma\inc\altera_avalon_sgdma_regs.h**,
- **<*install_dir*>\quartus\sopc_builder\components\altera_avalon_sgdma\HAL\inc\altera_avalon_sgdma.h**, and
- **<*install_dir*>\quartus\sopc_builder\components\altera_avalon_sgdma\HAL\inc\altera_avalon_sgdma_descriptor.h.**

Timeouts

The SG-DMA controller does not implement internal counters to detect stalls. Software can instantiate a timer component if this functionality is required.

SG-DMA Controller API

This section describes the application programming interface (API) for the SG-DMA controller core.

alt_avalon_sgdma_do_async_transfer()

Prototype:	`int alt_avalon_do_async_transfer(alt_sgdma_dev *dev, alt_sgdma_descriptor *desc)`
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`*dev`—a pointer to an SG-DMA device structure. `*desc`—a pointer to a single, constructed descriptor. The descriptor must have its "next" descriptor field initialized either to a non-ready descriptor, or to the next descriptor in the chain.
Returns:	Returns 0 success. Other return codes are defined in errno.h.
Description:	Set up and begin a non-blocking transfer of one or more descriptors or a descriptor chain. If the SG-DMA controller is busy at the time of this call, the routine will immediately return -EBUSY; the application can then decide how to proceed without being blocked. If a callback routine has been previously registered with this particular SG-DMA controller, the transfer will be set up to issue an interrupt on error, EOP, or chain completion. Otherwise, no interrupt is registered and it is the responsibility of the application developer to check for and handle errors and completion.

alt_avalon_sgdma_do_sync_transfer()

Prototype:	`alt_u8 alt_avalon_sgdma_do_sync_transfer(alt_sgdma_dev *dev, alt_sgdma_descriptor *desc)`
Thread-safe:	No.
Available from ISR:	Not recommended.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`*dev`—a pointer to an SG-DMA device structure. `*desc`—a pointer to a single, constructed descriptor. The descriptor must have its "next" descriptor field initialized either to a non-ready descriptor, or to the next descriptor in the chain.
Returns:	Returns the contents of the `status` register.
Description:	Sends a fully formed descriptor or list of descriptors to the SG-DMA controller for transfer. This function blocks both before transfer, if the SG-DMA controller is busy, and until the requested transfer has completed. If an error is detected during the transfer, it is abandoned and the controller's `status` register contents are returned to the caller. Additional error information is available in the status bits of each descriptor that the SG-DMA processed. It is the responsibility of the user application to search through the descriptor or list of descriptors to gather specific error information.

alt_avalon_sgdma_construct_mem_to_mem_desc()

Prototype: `void alt_avalon_sgdma_construct_mem_to_mem_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *read_addr, alt_u32 *write_addr, alt_u16 length, int read_fixed, int write_fixed)`

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters:
`*desc`—a pointer to the descriptor being constructed.
`*next`—a pointer to the "next" descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.
`*read_addr`—the first read address for the SG-DMA transfer.
`*write_addr`—the first write address for the SG-DMA transfer.
`length`—the number of bytes for the transfer.
`read_fixed`—if non-zero, the SG-DMA will read from a fixed address.
`write_fixed`—if non-zero, the SG-DMA will write to a fixed address.

Returns: `void`

Description: This function constructs a single SG-DMA descriptor in the memory specified in alt_avalon_sgdma_descriptor *desc for an Avalon-MM to Avalon-MM transfer. The function sets the OWNED_BY_HW bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the RUN bit of the SG-DMA `control` register is asserted.

The next field of the descriptor being constructed is set to the address in `*next`. The OWNED_BY_HW bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it will not process the descriptor at `*next` until its OWNED_BY_HW bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter.

You are responsible for properly allocating memory for the creation of both the descriptor under construction as well as the next descriptor in the chain.

Descriptors must be in a memory device mastered by the SG-DMA controller's chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller.

alt_avalon_sgdma_construct_stream_to_mem_desc()

Prototype:
```
void
alt_avalon_sgdma_construct_stream_to_mem_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *write_addr, alt_u16 length_or_eop, int write_fixed)
```

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters:
`*desc`—a pointer to the descriptor being constructed.
`*next`—a pointer to the "next" descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.
`*write_addr`—the first write address for the SG-DMA transfer.
`length_or_eop`—the number of bytes for the transfer. If set to zero (0x0), the transfer will continue until an EOP signal is received from the Avalon-ST interface.
`write_fixed`—if non-zero, the SG-DMA will write to a fixed address.

Returns: void

Description: This function constructs a single SG-DMA descriptor in the memory specified in alt_avalon_sgdma_descriptor `*desc` for an Avalon-ST to Avalon-MM transfer. The source (read) data for the transfer comes from the Avalon-ST interface connected to the SG-DMA controller's streaming read port.

The function sets the OWNED_BY_HW bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the RUN bit of the SG-DMA `control` register is asserted.

The next field of the descriptor being constructed is set to the address in `*next`. The OWNED_BY_HW bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it will not process the descriptor at `*next` until its OWNED_BY_HW bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter.

You are responsible for properly allocating memory for the creation of both the descriptor under construction as well as the next descriptor in the chain.

Descriptors must be in a memory device mastered by the SG-DMA controller's chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller.

alt_avalon_sgdma_construct_mem_to_stream_desc()

Prototype:
```
void
alt_avalon_sgdma_construct_mem_to_stream_desc(alt_sgdma_desc
riptor *desc, alt_sgdma_descriptor *next, alt_u32 *read_addr,
alt_u16 length, int read_fixed, int generate_sop, int
generate_eop, alt_u8 atlantic_channel)
```

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters:
`*desc`—a pointer to the descriptor being constructed.
`*next`—a pointer to the "next" descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.
`*read_addr`—the first read address for the SG-DMA transfer.
`length`—the number of bytes for the transfer.
`read_fixed`—if non-zero, the SG-DMA will read from a fixed address.
`generate_sop`—if non-zero, the SG-DMA will generate a start-of-packet (SOP) on the Avalon Streaming interface when commencing the transfer.
`generate_eop`—if non-zero, the SG-DMA will generate a end-of-packet (EOP) on the Avalon Streaming interface when completing the transfer.
`atlantic_channel`—an 8-bit channel identification number that will be passed to the Avalon-ST interface.

Returns: `void`

Description: This function constructs a single SG-DMA descriptor in the memory specified in alt_avalon_sgdma-descriptor `*desc` for an Avalon-MM to Avalon-ST transfer. The destination (write) data for the transfer goes to the Avalon-ST interface connected to the SG-DMA controller's streaming write port. The function sets the OWNED_BY_HW bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the RUN bit of the SG-DMA `control` register is asserted.

The next field of the descriptor being constructed is set to the address in `*next`. The OWNED_BY_HW bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it will not process the descriptor at `*next` until its OWNED_BY_HW bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter.

You are responsible for properly allocating memory for the creation of both the descriptor under construction as well as the next descriptor in the chain. Descriptors must be in a memory device mastered by the SG-DMA controller's chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller.

alt_avalon_sgdma_check_descriptor_status()

Prototype:	`int alt_avalon_sgdma_check_descriptor_status(alt_sgdma_descriptor *desc)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`*desc`—a pointer to the constructed descriptor to examine.
Returns:	Returns 0 if the descriptor is error-free, not owned by hardware, or a previously requested transfer completed normally. Other return codes are defined in **errno.h**.
Description:	Checks a descriptor previously owned by hardware for any errors reported in a previous transfer. The routine reports: errors reported by the SG-DMA controller, the buffer in use.

alt_avalon_sgdma_register_callback()

Prototype: `void alt_avalon_sgdma_register_callback(alt_sgdma_dev *dev, alt_avalon_sgdma_callback callback, alt_u16 chain_control, void *context)`

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*dev`—a pointer to the SG-DMA device structure.
`callback`—a pointer to the callback routine to execute at interrupt level.
`chain_control`—the SG-DMA control register contents.
`*context`—a pointer used to pass context-specific information to the ISR. `context` can point to any ISR-specific information.

Returns: `void`

Description: Associates a user-specific routine with the SG-DMA interrupt handler. If a callback is registered, all non-blocking transfers will enable interrupts that will cause the callback to be executed. The callback runs as part of the interrupt service routine, and great care must be taken to follow the guidelines for acceptable interrupt service routine behavior as described in the *Nios II Software Developer's Handbook*.

To disable callbacks after registering one, call this routine with 0x0 as the callback argument.

alt_avalon_sgdma_start()

Prototype:	`void alt_avalon_sgdma_start(alt_sgdma_dev *dev)`
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`*dev`—a pointer to the SG-DMA device structure.
Returns:	`void`
Description:	Starts the DMA engine and processes the descriptor pointed to in the controller's next descriptor pointer and all subsequent descriptors in the chain. It is not necessary to call this function when `do_sync` or `do_async` is used.

alt_avalon_sgdma_stop()

Prototype:	`void alt_avalon_sgdma_stop(alt_sgdma_dev *dev)`
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`*dev`—a pointer to the SG-DMA device structure.
Returns:	`void`
Description:	Stops the DMA engine following completion of the current buffer descriptor. It is not necessary to call this function when `do_sync` or `do_async` is used.

alt_avalon_sgdma_open()

Prototype:	`alt_sgdma_dev* alt_avalon_sgdma_open(const char* name)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`name`—the name of the SG-DMA device to open.
Returns:	A pointer to the SG-DMA device structure associated with the supplied name, or NULL if no corresponding SG-DMA device structure was found.
Description:	Retrieves a pointer to a hardware SG-DMA device structure.

Document Revision History

Table 6–13 shows the revision history for this chapter.

Table 6–13. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Chapter 6 was formerly Chapter 5. • Updated the description for the following sections and APIs: Instantiating the Core in SOPC Builder, Software Control, DMA Descriptors, alt_avalon_sgdma_start() and alt_avalon_sgdma_stop()	—
May 2007 v7.1.0	Initial release.	—

7. DMA Controller Core

NII51006-7.2.0

Core Overview

The direct memory access (DMA) controller core with Avalon® interface performs bulk data transfers, reading data from a source address range and writing the data to a different address range. An Avalon-MM master peripheral, such as a CPU, can offload memory transfer tasks to the DMA controller. While the DMA controller performs memory transfers, the master is free to perform other tasks in parallel.

The DMA controller transfers data as efficiently as possible, reading and writing data at the maximum pace allowed by the source or destination. The DMA controller is capable of performing Avalon transfers with flow control, enabling it to automatically transfer data to or from a slow peripheral with flow control (for example, a universal asynchronous receiver/transmitter [UART]), at the maximum pace allowed by the peripheral.

The DMA controller is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. For the Nios® II processor, device drivers are provided in the HAL system library. See "Software Programming Model" on page 7–6 for details of HAL support.

This chapter contains the following sections:

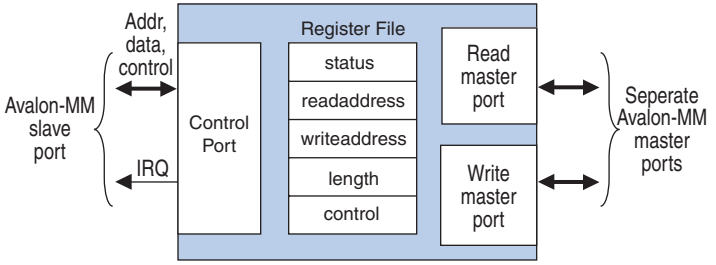
- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 7–4
- "Device and Tools Support" on page 7–6
- "Software Programming Model" on page 7–6

Functional Description

The DMA controller is used to perform direct memory-access data transfers from a source address-space to a destination address-space. The source and destination may be either an Avalon-MM slave peripheral (i.e., a constant address) or an address range in memory. The DMA controller can be used in conjunction with peripherals with flow control, which allows data transactions of fixed or variable length. The DMA controller can signal an interrupt request (IRQ) when a DMA transaction completes. A transaction is a sequence of one or more Avalon transfers initiated by the DMA controller core.

The DMA controller has two Avalon-MM master ports—a master read port and a master write port—and one Avalon-MM slave port for controlling the DMA as shown in Figure 7–1.

Figure 7–1. DMA Controller Block Diagram



A typical DMA transaction proceeds as follows:

1. A CPU prepares the DMA controller for a transaction by writing to the control port.

2. The CPU enables the DMA controller. The DMA controller then begins transferring data without additional intervention from the CPU. The DMA's master read port reads data from the read address, which may be a memory or a peripheral. The master write port writes the data to the destination address, which can also be a memory or peripheral. A shallow FIFO buffers data between the read and write ports.

3. The DMA transaction ends when a specified number of bytes are transferred (i.e., a fixed-length transaction), or an end-of-packet signal is asserted by either the sender or receiver (in other words, a variable-length transaction). At the end of the transaction, the DMA controller generates an interrupt request (IRQ) if it was configured by the CPU to do so.

4. During or after the transaction, the CPU can determine if a transaction is in progress, or if the transaction ended (and how) by examining the DMA controller's status register.

Setting Up DMA Transactions

An Avalon-MM master peripheral sets up and initiates DMA transactions by writing to registers via the control port. The Avalon-MM master programs the DMA engine using byte addresses which are byte aligned. The master peripheral configures the following options:

- Read (source) address location
- Write (destination) address location

- Size of the individual transfers: Byte (8-bit), halfword (16-bit), word (32-bit), doubleword (64-bit) or quadword (128-bit)
- Enable interrupt upon end of transaction
- Enable source or destination to end the DMA transaction with end-of-packet signal
- Specify whether source and destination are memory or peripheral

The master peripheral then sets a bit in the `control` register to initiate the DMA transaction.

The Master Read and Write Ports

The DMA controller reads data from the source address through the master read port, and then writes to the destination address through the master write port. The DMA controller is programmed using byte addresses. Read and write start addresses should be aligned to the transfer size. For example, to transfer data words, if the start address is 0, the address will increment to 4, 8 and 12. For heterogeneous systems where a number of different slave devices are of different widths, the data width for read and write masters matches the width of the widest data-width slave addressed by either the read or the write master. For bursting transfers, the burst length is set to the DMA transaction length with the appropriate unit conversion. For example, if a 32-bit data width DMA is programmed for a word transfer of 64 bytes, the length registered is programmed with 64 and the burst count port will be 16. If a 64-bit data width DMA is programmed for a doubleword transfer of 8 bytes, the length register is programmed with 8 and the burst count port will be 1.

There is a shallow FIFO buffer between the master read and write ports. The default depth is 2, which makes the write action depend on the data-available status of the FIFO, rather than on the status of the master read port.

Both the read and write master ports are capable of performing Avalon transfers with flow control, which allows the slave peripheral to control the flow of data and terminate the DMA transaction.



For details about flow control in Avalon-MM data transfers and Avalon-MM peripherals, refer to the *Avalon Memory-Mapped Interface Specification*.

Addressing and Address Incrementing

When accessing memory, the read (or write) address increments by 1, 2, 4, 8 or 16 after each access, depending on the width of the data. On the other hand, a typical peripheral device (such as UART) has fixed register locations. In this case, the read/write address is held constant throughout the DMA transaction.

The rules for address incrementing are, in order of priority:

- If the `control` register's `RCON` (or `WCON`) bit is set, the read (or write) increment value is 0.
- Otherwise, the read and write increment values are set according to the transfer size specified in the control register, as shown in Table 7–1.

Table 7–1. Address Increment Values

Transfer Width	Increment
byte	1
halfword	2
word	4
doubleword	8
quadword	16



In systems with heterogeneous data widths, care must be taken to present the correct address or offset when configuring the DMA to access native-aligned slaves. For example, in a system using a 32-bit Nios II processor and a 16-bit DMA, the base address for the UART `txdata` register must be divided by the `dma_data_width/cpu_data_width`—2 in this example.

Instantiating the Core in SOPC Builder

Use the MegaWizard® interface for the DMA controller in SOPC Builder to specify the core's configuration. Instantiating the DMA controller in SOPC Builder creates one slave port and two master ports. You must specify which slave peripherals can be accessed by the read and write master ports. Likewise, you must specify which other master peripheral(s) can access the DMA control port and initiate DMA transactions. The DMA controller does not export any signals to the top level of the system module.

DMA Parameters (Basic)

This section describes the parameters you can configure on the **DMA Parameters** page.

Transfer Size

The parameter **Width of the DMA Length Register** specifies the minimum width of the DMA's transaction length register, which can be between 1 and 32. The `length` register determines the maximum number of transfers possible in a single DMA transaction.

By default, the length register is wide enough to span any of the slave peripherals mastered by the read or write ports. Overriding the length register may be necessary if the DMA master port (read or write) masters only data peripherals, such as a UART. In this case, the address span of each slave is small, but a larger number of transfers may be desired per DMA transaction.

Burst Transactions

When **Enable Burst Transfers** is turned on, the DMA controller performs burst transactions on its master read and write ports. The parameter **Maximum Burst Size** determines the maximum burst size allowed in a transaction.

In burst mode, the length of a transaction must not be longer than the configured maximum burst size. Otherwise, the transaction must be performed as multiple transactions.

FIFO Implementation

This option determines the implementation of the FIFO buffer between the master read and write ports. Select **Construct FIFO from Registers** to implement the FIFO using one register per storage bit. This has a strong impact on logic utilization when the DMA controller's data width is large. See "Advanced Options" on page 7–6.

To implement the FIFO using embedded memory blocks available in the FPGA, select **Construct FIFO from Memory Blocks**.

Advanced Options

This section describes the parameters you can configure on the **Advanced Options** page.

Allowed Transactions

You can choose the transfer datawidth(s) supported by the DMA controller hardware. The following datawidth options can be enabled or disabled:

- Byte
- Halfword (two bytes)
- Word (four bytes)
- Doubleword (eight bytes)
- Quadword (sixteen bytes)

Disabling unnecessary transfer widths reduces the amount of on-chip logic resources consumed by the DMA controller core. For example, if a system has both 16-bit and 32-bit memories, but the DMA controller will only transfer data to the 16-bit memory, then 32-bit transfers could be disabled to conserve logic resources.

Device and Tools Support

The DMA Controller Core with Avalon Interface supports all Altera FPGA families.

Software Programming Model

This section describes the programming model for the DMA controller, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides HAL system library drivers that enable you to access the DMA controller core using the HAL API for DMA devices.

HAL System Library Support

The Altera-provided driver implements a HAL DMA device driver that integrates into the HAL system library for Nios II systems. HAL users should access the DMA controller via the familiar HAL API, rather than accessing the registers directly.



If your program uses the HAL device driver to access the DMA controller, accessing the device registers directly will interfere with the correct behavior of the driver.

The HAL DMA driver provides both ends of the DMA process; the driver registers itself as both a receive channel (alt_dma_rxchan) and a transmit channel (alt_dma_txchan). The *Nios II Software Developer's Handbook* provides complete details of the HAL system library and the usage of DMA devices.

ioctl() Operations

`ioctl()` operation requests are defined for both the receive and transmit channels, which allows you to control the hardware-dependent aspects of the DMA controller. Two `ioctl()` functions are defined for the receiver driver and the transmitter driver: `alt_dma_rxchan_ioctl()` and `alt_dma_txchan_ioctl()`. Table 7–2 lists the available operations. These are valid for both the transmit and receive channels.

Table 7–2. Operations for alt_dma_rxchan_ioctl() and alt_dma_txchan_ioctl()

Request	Meaning
`ALT_DMA_SET_MODE_8`	Transfers data in units of 8 bits. The value of "arg" is ignored.
`ALT_DMA_SET_MODE_16`	Transfers data in units of 16 bits. The value of "arg" is ignored.
`ALT_DMA_SET_MODE_32`	Transfers data in units of 32 bits. The value of "arg" is ignored.
`ALT_DMA_SET_MODE_64`	Transfers data in units of 64 bits. The value of "arg" is ignored.
`ALT_DMA_SET_MODE_128`	Transfers data in units of 128 bits. The value of "arg" is ignored.
`ALT_DMA_RX_ONLY_ON` *(1)*	Sets a DMA receiver into streaming mode. In this case, data is read continuously from a single location. The "arg" parameter specifies the address to read from.
`ALT_DMA_RX_ONLY_OFF` *(1)*	Turns off streaming mode for a receive channel. The value of "arg" is ignored.
`ALT_DMA_TX_ONLY_ON` *(1)*	Sets a DMA transmitter into streaming mode. In this case, data is written continuously to a single location. The "arg" parameter specifies the address to write to.
`ALT_DMA_TX_ONLY_OFF` *(1)*	Turns off streaming mode for a transmit channel. The value of "arg" is ignored.

Note to Table 7–2:

(1) These macro names changed in version 1.1 of the Nios II Embedded Design Suite (EDS). The old names (`ALT_DMA_TX_STREAM_ON`, `ALT_DMA_TX_STREAM_OFF`, `ALT_DMA_RX_STREAM_ON`, and `ALT_DMA_RX_STREAM_OFF`) are still valid, but new designs should use the new names.

Limitations

Currently the Altera-provided drivers do not support 64-bit and 128-bit DMA transactions.

This function is not thread safe. If you want to access the DMA controller from more than one thread then you should use a semaphore or mutex to ensure that only one thread is executing within this function at any time.

Software Files

The DMA controller is accompanied by the following software files. These files define the low-level interface to the hardware. Application developers should not modify these files.

- **altera_avalon_dma_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- **altera_avalon_dma.h**, **altera_avalon_dma.c**—These files implement the DMA controller's device driver for the HAL system library.

Register Map

Programmers using the HAL API never access the DMA controller hardware directly via its registers. In general, the register map is only useful to programmers writing a device driver.



The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate.

Table 7–3 shows the register map for the DMA controller. Device drivers control and communicate with the hardware through five memory-mapped 32-bit registers.

Table 7–3. DMA Controller Register Map

Offset	Register Name	Read/Write	31 . . 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	status *(1)*	RW	*(2)*									LEN	WEOP	REOP	BUSY	DONE
1	readaddress	RW	Read master start address													
2	writeaddress	RW	Write master start address													
3	length	RW	DMA transaction length (in bytes)													
4	—	—	Reserved *(3)*													
5	—	—	Reserved *(3)*													
6	control	RW	*(2)*	SOFTWARERESET	QUADWORD	DOUBLEWORD	WCON	RCON	LEEN	WEEN	REEN	I_EN	GO	WORD	HW	BYTE
7	—	—	Reserved *(3)*													

Notes to Table 7–3:

(1) Writing zero to the `status` register clears the LEN, WEOP, REOP, and DONE bits.
(2) These bits are reserved. Read values are undefined. Write zero.
(3) This register is reserved. Read values are undefined. The result of a write is undefined.

status Register

The `status` register consists of individual bits that indicate conditions inside the DMA controller. The `status` register can be read at any time. Reading the `status` register does not change its value.

The status register bits are shown in Table 7–4.

Table 7–4. status Register Bits

Bit Number	Bit Name	Read/Write/Clear	Description
0	DONE	R/C	A DMA transaction is completed. The DONE bit is set to 1 when an end of packet condition is detected or the specified transaction length is completed. Write zero to the status register to clear the DONE bit.
1	BUSY	R	The BUSY bit is 1 when a DMA transaction is in progress.
2	REOP	R	The REOP bit is 1 when a transaction is completed due to an end-of-packet event on the read side.
3	WEOP	R	The WEOP bit is 1 when a transaction is completed due to an end of packet event on the write side.
4	LEN	R	The LEN bit is set to 1 when the length register decrements to zero.

readaddress Register

The readaddress register specifies the first location to be read in a DMA transaction. The readaddress register width is determined at system generation time. It is wide enough to address the full range of all slave ports mastered by the read port.

writeaddress Register

The writeaddress register specifies the first location to be written in a DMA transaction. The writeaddress register width is determined at system generation time. It is wide enough to address the full range of all slave ports mastered by the write port.

length Register

The length register specifies the number of bytes to be transferred from the read port to the write port. The length register is specified in bytes. For example, the value must be a multiple of 4 for word transfers, and a multiple of 2 for halfword transfers.

The length register is decremented as each data value is written by the write master port. When length reaches 0 the LEN bit is set. The length register does not decrement below 0.

The length register width is determined at system generation time. It is at least wide enough to span any of the slave ports mastered by the read or write master ports, and it can be made wider if necessary.

control Register

The control register is composed of individual bits that control the DMA's internal operation. The control register's value can be read at any time. The control register bits determine which, if any, conditions of the DMA transaction result in the end of a transaction and an interrupt request.

The control register bits are shown in Table 7–5.

Table 7–5. control Register Bits (Part 1 of 2)

Bit Number	Bit Name	Read/Write/Clear	Description
0	BYTE	RW	Specifies byte transfers.
1	HW	RW	Specifies halfword (16-bit) transfers.
2	WORD	RW	Specifies word (32-bit) transfers.
3	GO	RW	Enables DMA transaction. When the GO bit is set to 0, the DMA is prevented from executing transfers. When the GO bit is set to 1 and the length register is non-zero, transfers occur.
4	I_EN	RW	Enables interrupt requests (IRQ). When the I_EN bit is 1, the DMA controller generates an IRQ when the status register's DONE bit is set to 1. IRQs are disabled when the I_EN bit is 0.
5	REEN	RW	Ends transaction on read-side end-of-packet. When the REEN bit is set to 1, a slave port with flow control on the read side may end the DMA transaction by asserting its end-of-packet signal.
6	WEEN	RW	Ends transaction on write-side end-of-packet. When the WEEN bit is set to 1, a slave port with flow control on the write side may end the DMA transaction by asserting its end-of-packet signal.
7	LEEN	RW	Ends transaction when the length register reaches zero. When the LEEN bit is 1, the DMA transaction ends when the length register reaches 0. When this bit is 0, length reaching 0 does not cause a transaction to end. In this case, the DMA transaction must be terminated by an end-of-packet signal from either the read or write master port.

Table 7–5. control Register Bits (Part 2 of 2)

Bit Number	Bit Name	Read/Write/Clear	Description
8	RCON	RW	Reads from a constant address. When RCON is 0, the read address increments after every data transfer. This is the mechanism for the DMA controller to read a range of memory addresses. When RCON is 1, the read address does not increment. This is the mechanism for the DMA controller to read from a peripheral at a constant memory address. For details, see "Addressing and Address Incrementing" on page 7–4.
9	WCON	RW	Writes to a constant address. Similar to the RCON bit, when WCON is 0 the write address increments after every data transfer; when WCON is 1 the write address does not increment. For details, see "Addressing and Address Incrementing" on page 7–4.
10	DOUBLEWORD	RW	Specifies doubleword transfers.
11	QUADWORD	RW	Specifies quadword transfers.
12	SOFTWARERESET	RW	Software can reset the DMA engine by writing this bit to 1 twice. Upon the second write of 1 to the SOFTWARERESET bit, the DMA control will be reset identically to a system reset. The logic which sequences the software reset process then resets itself automatically.

The data width of DMA transactions is specified by the BYTE, HW, WORD, DOUBLEWORD, and QUADWORD bits. Only one of these bits can be set at a time. If more than one of the bits is set, the DMA controller behavior is undefined. The width of the transfer is determined by the narrower of the two slaves read and written. For example, a DMA transaction that reads from a 16-bit flash memory and writes to a 32-bit on-chip memory requires a halfword transfer. In this case, HW must be set to 1, and BYTE, WORD, DOUBLEWORD, and QUADWORD must be set to 0.

To successfully perform transactions of a specific width, that width must be enabled in hardware using the **Allowed Transaction** hardware option. For example, the DMA controller behavior is undefined if quadword transfers are disabled in hardware, but the QUADWORD bit is set during a DMA transaction.



Executing a DMA software reset when a DMA transfer is active may result in permanent bus lockup (until the next system reset). The SOFTWARERESET bit should therefore not be written except as a last resort.

Interrupt Behavior

The DMA controller has a single IRQ output that is asserted when the `status` register's `DONE` bit equals 1 and the control register's `I_EN` bit equals 1.

Writing the `status` register clears the `DONE` bit and acknowledges the `IRQ`. A master peripheral can read the `status` register and determine how the DMA transaction finished by checking the `LEN`, `REOP`, and `WEOP` bits.

Referenced Document

This chapter references the Avalon Memory-Mapped Interface Specification manual.

Document Revision History

Table 7–6 shows the revision history for this chapter.

Table 7–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Chapter 7 was formerly Chapter 6. • Updated the description on Burst Transactions parameters.	—
May 2007 v7.1.0	• Chapter 6 was formerly Chapter 4. • Added "Device and Tools Support" on page 7–6 section. • Added note on addressing native-aligned peripherals. • Added table of contents to Overview section. • Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies. Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface." • Added description of `SOFTWARERESET` bit to `control` register in Table 4–5 on page 4–10. • Added more information about DMA addressing and the fact that addresses are aligned to the size of the data transfer in "The Master Read and Write Ports" on page 4–3.	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology. Other changes to the document serve only to clarify existing behavior.
May 2006 v6.0.0	Chapter title changed, but no change in content from previous release.	—
December 2005 v5.1.1	Changed Avalon "streaming" terminology to "flow control" based on a change to the *Avalon Interface Specification*	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
December 2004 v1.2	• Updated description of the `GO` bit. • Updated descriptions of `ioctl()` macros in table 6-2.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

Section II. Communication Peripherals

This section describes communication peripherals provided by Altera. These components provide communication interfaces for SOPC Builder systems.

See *About This Handbook* for further details.

This section includes the following chapters:

- Chapter 8, JTAG UART Core
- Chapter 9, UART Core
- Chapter 10, SPI Core



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

8. JTAG UART Core

NII51009-7.2.0

Core Overview

The JTAG universal asynchronous receiver/transmitter (UART) core with Avalon® interface implements a method to communicate serial character streams between a host PC and an SOPC Builder system on an Altera® FPGA. In many designs, the JTAG UART core eliminates the need for a separate RS-232 serial connection to a host PC for character I/O. The core provides a simple register-mapped Avalon interface that hides the complexities of the JTAG interface from embedded software programmers. Master peripherals (such as a Nios® II processor) communicate with the core by reading and writing control and data registers.

The JTAG UART core uses the JTAG circuitry built in to Altera FPGAs, and provides host access via the JTAG pins on the FPGA. The host PC can connect to the FPGA via any Altera JTAG download cable, such as the USB-Blaster™ cable. Software support for the JTAG UART core is provided by Altera. For the Nios II processor, device drivers are provided in the HAL system library, allowing software to access the core using the ANSI C Standard Library **stdio.h** routines. For the host PC, Altera provides JTAG terminal software that manages the connection to the target, decodes the JTAG data stream, and displays characters on screen.

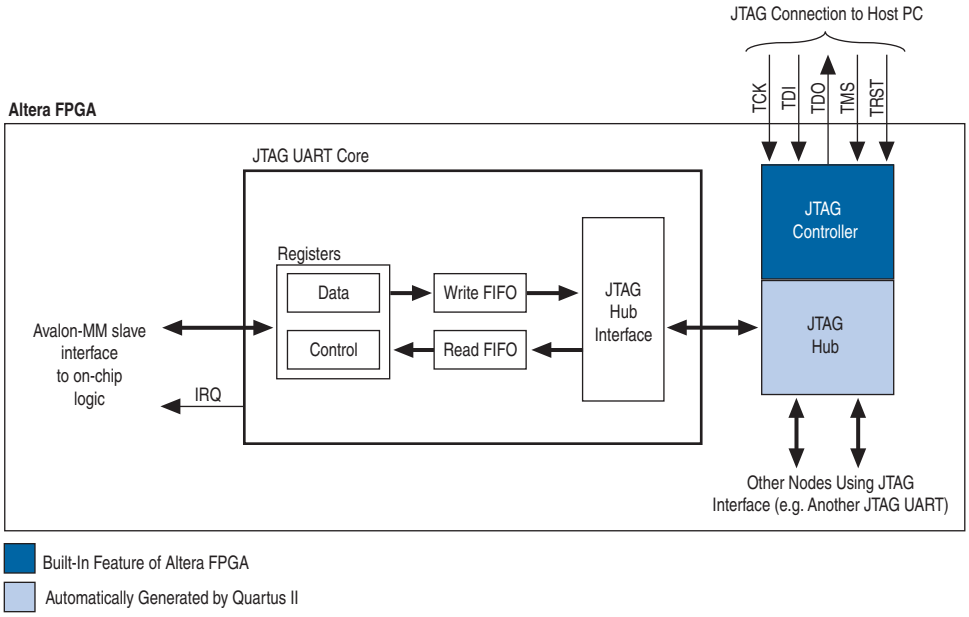
The JTAG UART core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 8–2
- "Device and Tools Support" on page 8–4
- "Instantiating the Core in SOPC Builder" on page 8–4
- "Hardware Simulation Considerations" on page 8–7
- "Software Programming Model" on page 8–7

Functional Description

Figure 8–1 shows a block diagram of the JTAG UART core and its connection to the JTAG circuitry inside an Altera FPGA. The following sections describe the components of the core.

Figure 8–1. JTAG UART Core Block Diagram



Avalon Slave Interface and Registers

The JTAG UART core provides an Avalon slave interface to the JTAG circuitry on an Altera FPGA. The user-visible interface to the JTAG UART core consists of two 32-bit registers, `data` and `control`, that are accessed through an Avalon slave port. An Avalon master, such as a Nios II processor, accesses the registers to control the core and transfer data over the JTAG connection. The core operates on 8-bit units of data at a time; eight bits of the `data` register serve as a one-character payload.

The JTAG UART core provides an active-high interrupt output that can request an interrupt when read data is available, or when the write FIFO is ready for data. For further details see "Interrupt Behavior" on page 8–13.

Read and Write FIFOs

The JTAG UART core provides bidirectional FIFOs to improve bandwidth over the JTAG connection. The FIFO depth is parameterizable to accommodate the available on-chip memory. The FIFOs can be constructed out of memory blocks or registers, allowing you to trade off logic resources for memory resources, if necessary.

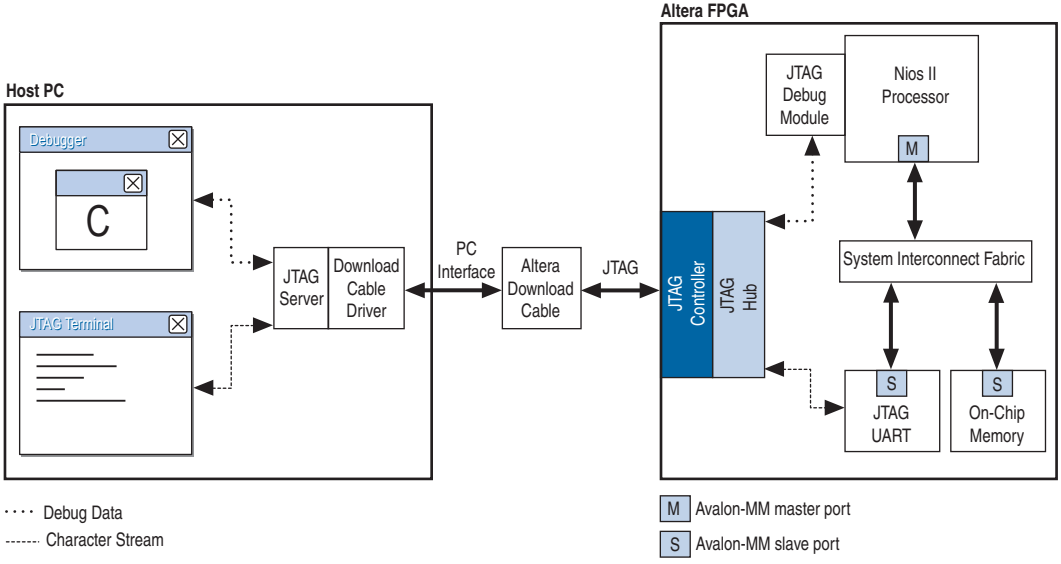
JTAG Interface

Altera FPGAs contain built-in JTAG control circuitry between the device's JTAG pins and the logic inside the device. The JTAG controller can connect to user-defined circuits called "nodes" implemented in the FPGA. Because several nodes may need to communicate via the JTAG interface, a JTAG hub (that is, a multiplexer) is necessary. During logic synthesis and fitting, the Quartus® II software automatically generates the JTAG hub logic. No manual design effort is required to connect the JTAG circuitry inside the device; the process is presented here only for clarity.

Host-Target Connection

Figure 8–2 shows the connection between a host PC and an SOPC Builder-generated system containing a JTAG UART core.

Figure 8–2. Example System Using the JTAG UART Core



The JTAG controller on the FPGA and the download cable driver on the host PC implement a simple data-link layer between host and target. All JTAG nodes inside the FPGA are multiplexed through the single JTAG connection. JTAG server software on the host PC controls and decodes the JTAG data stream, and maintains distinct connections with nodes inside the FPGA.

The example system in Figure 8–2 contains one JTAG UART core and a Nios II processor. Both agents communicate with the host PC over a single Altera download cable. Thanks to the JTAG server software, each host application has an independent connection to the target. Altera provides the JTAG server drivers and host software required to communicate with the JTAG UART core.

Systems with multiple JTAG UART cores are possible, and all cores communicate via the same JTAG interface. To maintain coherent data streams, only one processor should communicate with each JTAG UART core.

Device and Tools Support

The JTAG UART core supports the Arria™ GX, Stratix® III, Stratix II, Stratix II GX, Stratix GX, Stratix, Cyclone® III, Cyclone II, and Cyclone device families. The JTAG UART core is supported by the Nios II hardware abstraction layer (HAL) system library. No software support is provided for the first-generation Nios processor.

To view the character stream on the host PC, the JTAG UART core must be used in conjunction with the JTAG terminal software provided by Altera. Nios II processor users access the JTAG UART via the Nios II IDE or the **nios2-terminal** command-line utility.

For further details, refer to the *Nios II Software Developer's Handbook* or the Nios II IDE online help

Instantiating the Core in SOPC Builder

Designers use the MegaWizard® interface for the JTAG UART core in SOPC Builder to specify the core features. The following sections describe the available options in the MegaWizard interface.

Configuration Page

The options on this page control the hardware configuration of the JTAG UART core. The default settings are pre-configured to behave optimally with the Altera-provided device drivers and JTAG terminal software. Most designers should not change the default values, except for the **Construct using registers instead of memory blocks** option.

Write FIFO Settings

The write FIFO buffers data flowing from the Avalon interface to the host. The following settings are available:

- **Depth**—The write FIFO depth can be set from 8 to 32,768 bytes. Only powers of two are allowed. Larger values consume more on-chip memory resources. A depth of 64 is generally optimal for performance, and larger values are rarely necessary.
- **IRQ Threshold**—The write IRQ threshold governs how the core asserts its IRQ in response to the FIFO emptying. As the JTAG circuitry empties data from the write FIFO, the core asserts its IRQ when the number of characters remaining in the FIFO reaches this threshold value. For maximum bandwidth, a processor should service the interrupt by writing more data and preventing the write FIFO from emptying completely. A value of 8 is typically optimal. See "Interrupt Behavior" on page 8–13 for further details.
- **Construct using registers instead of memory blocks**—Turning on this option causes the FIFO to be constructed out of on-chip logic resources. This option is useful when memory resources are limited. Each byte consumes roughly 11 logic elements (LEs), so a FIFO depth of 8 (bytes) consumes roughly 88 LEs.

Read FIFO Settings

The read FIFO buffers data flowing from the host to the Avalon interface. Settings are available to control the depth of the FIFO and the generation of interrupts.

- **Depth**—The read FIFO depth can be set from 8 to 32,768 bytes. Only powers of two are allowed. Larger values consume more on-chip memory resources. A depth of 64 is generally optimal for performance, and larger values are rarely necessary.
- **IRQ Threshold—**The IRQ threshold governs how the core asserts its IRQ in response to the FIFO filling up. As the JTAG circuitry fills up the read FIFO, the core asserts its IRQ when the amount of space remaining in the FIFO reaches this threshold value. For maximum bandwidth, a processor should service the interrupt by reading data and preventing the read FIFO from filling up completely. A value of 8 is typically optimal. See "Interrupt Behavior" on page 8–13 for further details.

- **Construct using registers instead of memory blocks**—Turning on this option causes the FIFO to be constructed out of logic resources. This option is useful when memory resources are limited. Each byte consumes roughly 11 LEs, so a FIFO depth of 8 (bytes) consumes roughly 88 LEs.

Simulation Settings

At system generation time, when SOPC Builder generates the logic for the JTAG UART core, a simulation model is also constructed. The simulation model offers features to simplify simulation of systems using the JTAG UART core. Changes to the simulation settings do not affect the behavior of the core in hardware; the settings affect only functional simulation.

Simulated Input Character Stream

You can enter a character stream that will be simulated entering the read FIFO upon simulated system reset. The MegaWizard interface accepts an arbitrary character string, which is later incorporated into the test bench. After reset, this character string is pre-initialized in the read FIFO, giving the appearance that an external JTAG terminal program is sending a character stream to the JTAG UART core.

Prepare Interactive Windows

At system generation time, the JTAG UART core generator can create ModelSim® macros to open interactive windows during simulation. These windows allow the user to send and receive ASCII characters via a console, giving the appearance of a terminal session with the system executing in hardware. The following options are available:

- **Do not generate ModelSim aliases for interactive windows**—This option does not create any ModelSim macros for character I/O.
- **Create ModelSim alias to open a window showing output as ASCII text**—This option creates a ModelSim macro to open a console window that displays output from the write FIFO. Values written to the write FIFO via the Avalon interface are displayed in the console as ASCII characters.
- **Create ModelSim alias to open an interactive stimulus/response window**—This option creates a ModelSim macro to open a console window that allows input and output interaction with the core. Values written to the write FIFO via the Avalon interface are displayed in the console as ASCII characters. Characters typed into

the console are fed into the read FIFO, and can be read via the Avalon interface. When this option is enabled, the simulated character input stream option is ignored.

Hardware Simulation Considerations

The simulation features were created for easy simulation of Nios II processor systems when using the ModelSim simulator. The simulation model is implemented in the JTAG UART core's top-level HDL file. The synthesizable HDL and the simulation HDL are implemented in the same file. Some simulation features are implemented using "translate on/off" synthesis directives that make certain sections of HDL code visible only to the synthesis tool.



Refer to *AN 351: Simulating Nios II Processor Designs* for complete details about simulating the JTAG UART core in Nios II systems.

Other simulators can be used, but require user effort to create a custom simulation process. You can use the auto-generated ModelSim scripts as references to create similar functionality for other simulators.



Do not edit the simulation directives if you are using Altera's recommended simulation procedures. If you change the simulation directives to create a custom simulation flow, be aware that SOPC Builder overwrites existing files during system generation. Take precautions to ensure your changes are not overwritten.

Software Programming Model

The following sections describe the software programming model for the JTAG UART core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides HAL system library drivers that enable you to access the JTAG UART using the ANSI C standard library functions, such as `printf()` and `getchar()`.

HAL System Library Support

The Altera-provided driver implements a HAL character-mode device driver that integrates into the HAL system library for Nios II systems. HAL users should access the JTAG UART via the familiar HAL API and the ANSI C standard library, rather than accessing the JTAG UART registers. `ioctl()` requests are defined that allow HAL users to control the hardware-dependent aspects of the JTAG UART.



If your program uses the Altera-provided HAL device driver to access the JTAG UART hardware, accessing the device registers directly will interfere with the correct behavior of the driver.

For Nios II processor users, the HAL system library API provides complete access to the JTAG UART core's features. Nios II programs treat the JTAG UART core as a character mode device, and send and receive data using the ANSI C standard library functions, such as `getchar()` and `printf()`.

Example 8–1 demonstrates the simplest possible usage, printing a message to stdout using `printf()`. In this example, the SOPC Builder system contains a JTAG UART core, and the HAL system library is configured to use this JTAG UART device for stdout.

Example 8–1. Printing Characters to a JTAG UART Core as stdout

```
#include <stdio.h>
int main ()
{
  printf("Hello world.\n");
  return 0;
}
```

Example 8–2 demonstrates reading characters from and sending messages to a JTAG UART core using the C standard library. In this example, the SOPC Builder system contains a JTAG UART core named `jtag_uart` that is not necessarily configured as the stdout device. In this case, the program treats the device like any other node in the HAL file system.

Example 8–2. Transmitting Characters to a JTAG UART Core

```
/* A simple program that recognizes the characters 't' and 'v' */
#include <stdio.h>
#include <string.h>
int main ()
{
  char* msg = "Detected the character 't'.\n";
  FILE* fp;
  char prompt = 0;

  fp = fopen ("/dev/jtag_uart", "r+"); //Open file for reading and writing
  if (fp)
  {
    while (prompt != 'v')
    {  // Loop until we receive a 'v'.
      prompt = getc(fp);  // Get a character from the JTAG UART.
      if (prompt == 't')
      {  // Print a message if character is 't'.
        fwrite (msg, strlen (msg), 1, fp);
      }

      if (ferror(fp))// Check if an error occurred with the file pointer
        clearerr(fp);// If so, clear it.
    }

    fprintf(fp, "Closing the JTAG UART file handle.\n");
    fclose (fp);
  }

  return 0;
}
```

In this example, the `ferror(fp)` is used to check if an error occurred on the JTAG UART connection, such as a disconnected JTAG connection. In this case, the driver detects that the JTAG connection is disconnected, reports an error (`EIO`), and discards data for subsequent transactions. If this error ever occurs, the C library latches the value until you explicitly clear it with the `clearerr()` function.

The *Nios II Software Developer's Handbook* provides complete details of the HAL system library. The Nios II Embedded Design Suite (EDS) provides a number of software example designs that use the JTAG UART core.

Driver Options: Fast versus Small Implementations

To accommodate the requirements of different types of systems, the JTAG UART driver has two variants, a fast version and a small version. The fast behavior is used by default. Both the fast and small drivers fully support the C standard library functions and the HAL API.

The fast driver is an interrupt-driven implementation, which allows the processor to perform other tasks when the device is not ready to send or receive data. Because the JTAG UART data rate is slow compared to the processor, the fast driver can provide a large performance benefit for systems that could be performing other tasks in the interim. In addition, the fast version of the Altera Avalon JTAG UART monitors the connection to the host. The driver discards characters if no host is connected, or if the host is not running an application that handles the I/O stream.

The small driver is a polled implementation that waits for the JTAG UART hardware before sending and receiving each character. The performance of the small driver is poor if you are sending large amounts of data. The small version assumes that the host is always connected, and will never discard characters. Therefore, the small driver will hang the system if the JTAG UART hardware is ever disconnected from the host while the program is sending or receiving data. There are two ways to enable the small footprint driver:

- Enable the small footprint setting for the HAL system library project. This option affects device drivers for all devices in the system.
- Specify the preprocessor option `-DALTERA_AVALON_JTAG_UART_SMALL`. Use this option if you want the small, polled implementation of the JTAG UART driver, but you do not want to affect the drivers for other devices.

ioctl() Operations

The fast version of the JTAG UART driver supports the `ioctl()` function to allow HAL-based programs to request device-specific operations. Specifically, you can use the `ioctl()` operations to control the timeout period, and to detect whether or not a host is connected. The fast driver defines the `ioctl()` operations shown in Table 8–1.

Table 8–1. JTAG UART ioctl() Operations for the Fast Driver Only

Request	Meaning
`TIOCSTIMEOUT`	Set the timeout (in seconds) after which the driver will decide that the host is not connected. A timeout of 0 makes the target assume that the host is always connected. The `ioctl` arg parameter passed in must be a pointer to an integer.
`TIOCGCONNECTED`	Sets the integer arg parameter to a value that indicates whether the host is connected and acting as a terminal (1), or not connected (0). The `ioctl` arg parameter passed in must be a pointer to an integer.

For details about the `ioctl()` function, refer to the *Nios II Software Developer's Handbook*.

Software Files

The JTAG UART core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_jtag_uart_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- **altera_avalon_jtag_uart.h**, **altera_avalon_jtag_uart.c**—These files implement the HAL system library device driver.

Accessing the JTAG UART Core via a Host PC

Host software is necessary for a PC to access the JTAG UART core. The Nios II IDE supports the JTAG UART core, and displays character I/O in a console window. Altera also provides a command-line utility called **nios2-terminal** that opens a terminal session with the JTAG UART core.

For further details, refer to the *Nios II Software Developer's Handbook* and the Nios II IDE online help.

Register Map

Programmers using the HAL API never access the JTAG UART core directly via its registers. In general, the register map is only useful to programmers writing a device driver for the core.

CAUTION

The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate.

Table 8–2 shows the register map for the JTAG UART core. Device drivers control and communicate with the core through the two 32-bit memory-mapped registers.

Table 8–2. JTAG UART Core Register Map

Offset	Register Name	R/W	Bit Description													
			31	…	16	15	14	…	11	10	9	8	7	…	2	1	0
0	data	RW	RAVAIL			RVALID	*(1)*						DATA				
1	control	RW	WSPACE			*(1)*				AC	WI	RI	*(1)*			WE	RE

Note to Table 8–2:
(1) Reserved. Read values are undefined. Write zero.

Data Register

Embedded software accesses the read and write FIFOs via the `data` register. Table 8–3 describes the function of each bit.

Table 8–3. data Register Bits

Bit Number	Bit/Field Name	Read/Write/Clear	Description
0 .. 7	DATA	R/W	The value to transfer to/from the JTAG core. When writing, the DATA field holds a character to be written to the write FIFO. When reading, the DATA field holds a character read from the read FIFO.
15	RVALID	R	Indicates whether the DATA field is valid. If RVALID=1, then the DATA field is valid, otherwise DATA is undefined.
16 .. 32	RAVAIL	R	The number of characters remaining in the read FIFO (after the current read).

A read from the `data` register returns the first character from the FIFO (if one is available) in the DATA field. Reading also returns information about the number of characters remaining in the FIFO in the RAVAIL field. A write to the `data` register stores the value of the DATA field in the write FIFO. If the write FIFO is full, then the character is lost.

Control Register

Embedded software controls the JTAG UART core's interrupt generation and reads status information via the `control` register. Table 8–4 describes the function of each bit.

Table 8–4. control Register Bits

Bit Number	Bit/Field Name	Read/Write/Clear	Description
0	RE	R/W	Interrupt-enable bit for read interrupts
1	WE	R/W	Interrupt-enable bit for write interrupts
8	RI	R	Indicates that the read interrupt is pending
9	WI	R	Indicates that the write interrupt is pending
10	AC	R/C	Indicates that there has been JTAG activity since the bit was cleared. Writing 1 to AC clears it to 0.
16 .. 32	WSPACE	R	The number of spaces available in the write FIFO.

A read from the `control` register returns the status of the `read` and `write` FIFOs. Writes to the register can be used to enable/disable interrupts, or clear the `AC` bit.

The `RE` and `WE` bits enable interrupts for the read and write FIFOs, respectively. The `WI` and `RI` bits indicate the status of the interrupt sources, qualified by the values of the interrupt enable bits (`WE` and `RE`). Embedded software can examine `RI` and `WI` to determine the condition that generated the IRQ. See "Interrupt Behavior" on page 8–13 for further details.

The `AC` bit indicates that an application on the host PC has polled the JTAG UART core via the JTAG interface. Once set, the `AC` bit remains set until it is explicitly cleared via the Avalon interface. Writing `1` to `AC` clears it. Embedded software can examine the `AC` bit to determine if a connection exists to a host PC. If no connection exists, the software may choose to ignore the JTAG data stream. When the host PC has no data to transfer, it can choose to poll the JTAG UART core as infrequently as once per second. Delays caused by other host software using the JTAG download cable could cause delays of up to 10 seconds between polls.

Interrupt Behavior

The JTAG UART core generates an interrupt when either of the individual interrupt conditions is pending and enabled.



Interrupt behavior is of interest to device driver programmers concerned with the bandwidth performance to the host PC. Example designs and the JTAG terminal program provided with Nios II Embedded Design Suite (EDS) are pre-configured with optimal interrupt behavior.

The JTAG UART core has two kinds of interrupts: write interrupts and read interrupts. The WE and RE bits in the `control` register enable/disable the interrupts.

The core can assert a write interrupt whenever the write FIFO is nearly empty. The "nearly empty" threshold, `write_threshold`, is specified at system generation time and cannot be changed by embedded software. The write interrupt condition is set whenever there are `write_threshold` or fewer characters in the write FIFO. It is cleared by writing characters to fill the write FIFO beyond the `write_threshold`. Embedded software should only enable write interrupts after filling the write FIFO. If it has no characters remaining to send, embedded software should disable the write interrupt.

The core can assert a read interrupt whenever the read FIFO is nearly full. The "nearly full" threshold value, `read_threshold`, is specified at system generation time and cannot be changed by embedded software. The read interrupt condition is set whenever the read FIFO has `read_threshold` or fewer spaces remaining. The read interrupt condition is also set if there is at least one character in the read FIFO and no more characters are expected. The read interrupt is cleared by reading characters from the read FIFO.

For optimum performance, the interrupt thresholds should match the interrupt response time of the embedded software. For example, with a 10-MHz JTAG clock, a new character is provided (or consumed) by the host PC every 1µs. With a threshold of 8, the interrupt response time must be less than 8µs. If the interrupt response time is too long, then performance will suffer. If it is too short, then interrupts will occur too frequently.



For Nios II processor systems, read and write thresholds of 8 are an appropriate default.

Referenced Document

This chapter references the *Nios II Software Developer's Handbook*.

Document Revision History

Table 8–5 shows the revision history for this chapter.

Table 8–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 8 was formerly Chapter 7.	—
May 2007 v7.1.0	• Chapter 7 was formerly chapter 5. • Added Arria™ GX to "Device and Tools Support" on page 8–4. • Added table of contents to Overview section. • Added Referenced Documents section.	—
March 2007 v7.0.0	Added Cyclone III and Stratix III support.	Version 7.0 of the Quartus II software added Cyclone III support.
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies. • Changed old "Avalon switch fabric" term to "system interconnect fabric." • Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface."	For version 6.1, added Stratix III support. Additionally, Altera released the Avalon Streaming interface, which necessitated some rephrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
December 2004 v1.2	Added Cyclone II support.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

9. UART Core

NII51010-7.2.0

Core Overview

The universal asynchronous receiver/transmitter core with Avalon® interface (UART core) implements a method to communicate serial character streams between an embedded system on an Altera® FPGA and an external device. The core implements the RS-232 protocol timing, and provides adjustable baud rate, parity, stop and data bits, and optional `RTS`/`CTS` flow control signals. The feature set is configurable, allowing designers to implement just the necessary functionality for a given system.

The core provides a simple register-mapped Avalon Memory-Mapped (Avalon-MM) slave interface that allows Avalon-MM master peripherals (such as a Nios® II processor) to communicate with the core simply by reading and writing control and data registers.

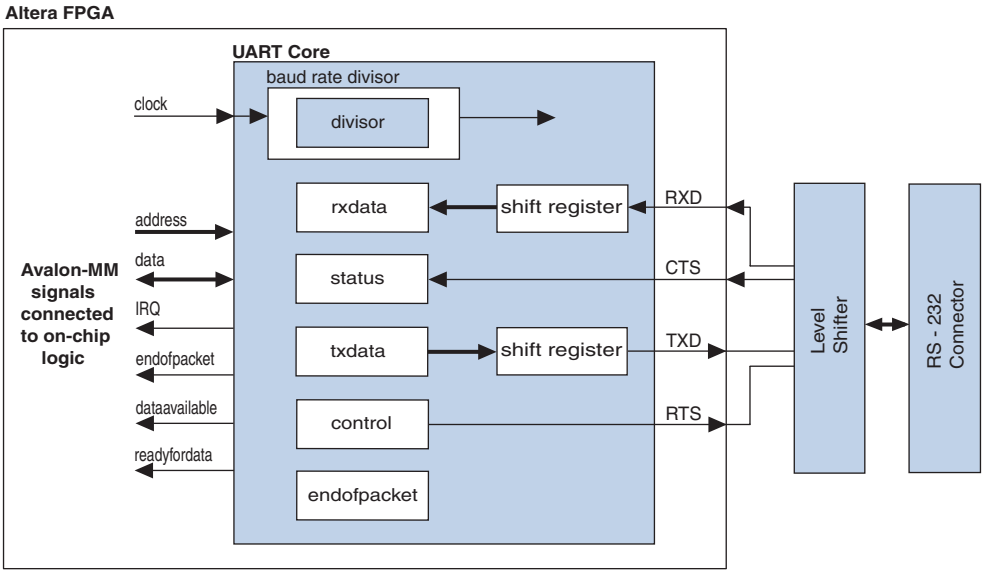
The UART core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 9–2
- "Device and Tools Support" on page 9–4
- "Instantiating the Core in SOPC Builder" on page 9–5
- "Hardware Simulation Considerations" on page 9–9
- "Software Programming Model" on page 9–9

Functional Description

Figure 9–1 shows a block diagram of the UART core.

Figure 9–1. Block Diagram of the UART Core in a Typical System



The core has two user-visible parts:

- The register file, which is accessed via the Avalon-MM slave port
- The RS-232 signals, `RXD`, `TXD`, `CTS`, and `RTS`

Avalon-MM Slave Interface and Registers

The UART core provides an Avalon-MM slave interface to the internal register file. The user interface to the UART core consists of six 16-bit registers: `control`, `status`, `rxdata`, `txdata`, `divisor`, and `endofpacket`. A master peripheral, such as a Nios II processor, accesses the registers to control the core and transfer data over the serial connection.

The UART core provides an active-high interrupt request (IRQ) output that can request an interrupt when new data has been received, or when the core is ready to transmit another character. For further details, refer "Interrupt Behavior" on page 9–20.

The Avalon-MM slave port is capable of transfers with flow control. The UART core can be used in conjunction with a direct memory access (DMA) peripheral with Avalon-MM flow control to automate continuous data transfers between, for example, the UART core and memory.



For more information, refer to the *Timer Core* chapter in volume 5 of the *Quartus II Handbook*. For details about the Avalon-MM interface, refer to the *Avalon Memory-Mapped Interface Specification*.

RS-232 Interface

The UART core implements RS-232 asynchronous transmit and receive logic. The UART core sends and receives serial data via the TXD and RXD ports. The I/O buffers on most Altera FPGA families do not comply with RS-232 voltage levels, and may be damaged if driven directly by signals from an RS-232 connector. To comply with RS-232 voltage signaling specifications, an external level-shifting buffer is required (for example, Maxim MAX3237) between the FPGA I/O pins and the external RS-232 connector.

The UART core uses a logic 0 for mark, and a logic 1 for space. An inverter inside the FPGA can be used to reverse the polarity of any of the RS-232 signals, if necessary.

Transmitter Logic

The UART transmitter consists of a 7-, 8-, or 9-bit `txdata` holding register and a corresponding 7-, 8-, or 9-bit transmit shift register. Avalon-MM master peripherals write the `txdata` holding register via the Avalon-MM slave port. The transmit shift register is loaded from the `txdata` register automatically when a serial transmit shift operation is not currently in progress. The transmit shift register directly feeds the TXD output. Data is shifted out to TXD least-significant bit (LSB) first.

These two registers provide double buffering. A master peripheral can write a new value into the `txdata` register while the previously written character is being shifted out. The master peripheral can monitor the transmitter's status by reading the `status` register's transmitter ready (`TRDY`), transmitter shift register empty (`tmt`), and transmitter overrun error (`toe`) bits.

The transmitter logic automatically inserts the correct number of start, stop, and parity bits in the serial TXD data stream as required by the RS-232 specification.

Receiver Logic

The UART receiver consists of a 7-, 8-, or 9-bit receiver-shift register and a corresponding 7-, 8-, or 9-bit `rxdata` holding register. Avalon-MM master peripherals read the `rxdata` holding register via the Avalon-MM slave port. The `rxdata` holding register is loaded from the receiver shift register automatically every time a new character is fully received.

These two registers provide double buffering. The `rxdata` register can hold a previously received character while the subsequent character is being shifted into the receiver shift register.

A master peripheral can monitor the receiver's status by reading the `status` register's read-ready (`rrdy`), receiver-overrun error (`roe`), break detect (`BRK`), parity error (`pe`), and framing error (`fe`) bits. The receiver logic automatically detects the correct number of start, stop, and parity bits in the serial RXD stream as required by the RS-232 specification. The receiver logic checks for four exceptional conditions in the received data (frame error, parity error, receive overrun error, and break), and sets corresponding status register bits (`fe`, `pe`, `roe`, or `BRK`).

Baud Rate Generation

The UART core's internal baud clock is derived from the Avalon-MM clock input. The internal baud clock is generated by a clock divider. The divisor value can come from one of the following sources:

- A constant value specified at system generation time
- The 16-bit value stored in the `divisor` register

The `divisor` register is an optional hardware feature. If it is disabled at system generation time, the divisor value is fixed, and the baud rate cannot be altered.

Device and Tools Support

The UART core can target all Altera FPGAs.

Instantiating the Core in SOPC Builder

Instantiating the UART in hardware creates at least two I/O ports for each UART core: An `RXD` input, and a `TXD` output. Optionally, the hardware may include flow control signals, the `CTS` input and RTS output.

Designers use the MegaWizard® interface for the UART core in SOPC Builder to configure the hardware feature set. The following sections describe the available options.

Configuration Settings

This section describes the configuration settings.

Baud Rate Options

The UART core can implement any of the standard baud rates for RS-232 connections. The baud rate can be configured in one of two ways:

- **Fixed rate**—The baud rate is fixed at system generation time and cannot be changed via the Avalon-MM slave port.
- **Variable rate**—The baud rate can vary, based on a clock divisor value held in the `divisor` register. A master peripheral changes the baud rate by writing new values to the `divisor` register.



The baud rate is calculated based on the clock frequency provided by the Avalon-MM interface. Changing the system clock frequency in hardware without re-generating the UART core hardware will result in incorrect signaling.

Baud Rate (bps) Setting
The **Baud Rate** setting determines the default baud rate after reset. The **Baud Rate** option offers standard preset values (for example, 9600, 57600, 115200 bps), or you can enter any baud rate manually.

The baud rate value is used to calculate an appropriate clock divisor value to implement the desired baud rate. Baud rate and divisor values are related as follows:

(1) $$\text{divisor} = \frac{int(\text{clock frequency})}{\text{baud rate}} + 0.5$$

(2) $$\text{baud rate} = \frac{\text{clock frequency}}{(\text{divisor} + 1)}$$

Baud Rate Can Be Changed By Software Setting
When this setting is on, the hardware includes a 16-bit `divisor` register at address offset 4. The `divisor` register is writable, so the baud rate can be changed by writing a new value to this register.

When this setting is off, the UART hardware does not include a `divisor` register. The UART hardware implements a constant (unchangeable) baud divisor, and the value cannot be changed after system generation. In this case, writing to address offset 4 has no effect, and reading from address offset 4 produces an undefined result.

Data Bits, Stop Bits, Parity

The UART core's parity, data bits and stop bits are configurable. These settings are fixed at system generation time; they cannot be altered via the register file. The following settings are available.

Data Bits Setting
The settings shown in Table 9–1 are available.

Table 9–1. Data Bits Settings

Setting	Allowed Values	Description
Data Bits	7, 8, 9	This setting determines the widths of the `txdata`, `rxdata`, and `endofpacket` registers.
Stop Bits	1, 2	This setting determines whether the core transmits 1 or 2 stop bits with every character. The core always terminates a receive transaction at the first stop bit, and ignores all subsequent stop bits, regardless of the Stop Bits setting.
Parity	None, Even, Odd	This setting determines whether the UART transmits characters with parity checking, and whether it expects received characters to have parity checking. Refer to "Parity Setting".

Parity Setting
When **Parity** is set to **None**, the transmit logic sends data without including a parity bit, and the receive logic presumes the incoming data does not include a parity bit. When parity is **None**, the status register's parity error (PE) bit is not implemented; it always reads 0.

When **Parity** is set to **Odd** or **Even**, the transmit logic computes and inserts the required parity bit into the outgoing `TXD` bitstream, and the receive logic checks the parity bit in the incoming `RXD` bitstream. If the receiver finds data with incorrect parity, the status register's PE is set to 1. When parity is **Even**, the parity bit is 0 if the character has an even number of 1 bits; otherwise the parity bit is 1. Similarly, when parity is **Odd**, the parity bit is 0 if the character has an odd number of 1 bits.

Flow Control

The following flow control option is available.

Include CTS/RTS Pins and Control Register Bits
When this setting is on, the UART hardware includes:

- `cts_n` (logic negative CTS) input port
- `rts_n` (logic negative RTS) output port
- CTS bit in the `status` register
- DCTS bit in the `status` register
- RTS bit in the `control` register
- IDCTS bit in the `control` register

Based on these hardware facilities, an Avalon-MM master peripheral can detect CTS and transmit RTS flow control signals. The CTS input and RTS output ports are tied directly to bits in the `status` and `control` registers, and have no direct effect on any other part of the core. When using flow control, be sure the terminal program on the host side is also configured for flow control.

When the **Include CTS/RTS pins and control register bits** setting is off, the core does not include the hardware listed above and continuous writes to the UART may loose data. The control/status bits CTS, DCTS, IDCTS, and RTS are not implemented; they always read as 0.

Avalon-MM Transfers with Flow Control (DMA)

The UART core's Avalon-MM interface optionally implements Avalon-MM transfers with flow control. This allows an Avalon-MM master peripheral to write data only when the UART core is ready to accept another character, and to read data only when the core has data available. The UART core can also optionally include the end-of-packet register.

Include End-of-Packet Register
When this setting is on, the UART core includes:

- A 7-, 8-, or 9-bit `endofpacket` register at address-offset 5. The data width is determined by the **Data Bits** setting.
- eop bit in the status register
- ieop bit in the control register
- `endofpacket` signal in the Avalon-MM interface to support data transfers with flow control to/from other master peripherals in the system

End-of-packet (EOP) detection allows the UART core to terminate a data transaction with a Avalon-MM master with flow control. EOP detection can be used with a DMA controller, for example, to implement a UART that automatically writes received characters to memory until a specified character is encountered in the incoming RXD stream. The terminating (EOP) character's value is determined by the endofpacket register.

When the EOP register is disabled, the UART core does not include the resources listed above. Writing to the endofpacket register has no effect, and reading produces an undefined value.

Simulation Settings

When the UART core's logic is generated, a simulation model is also constructed. The simulation model offers features to simplify and accelerate simulation of systems that use the UART core. Changes to the simulation settings do not affect the behavior of the UART core in hardware; the settings affect only functional simulation.



For examples of how to use the following settings to simulate Nios II systems, refer to *AN 351: Simulating Nios II Embedded Processor Designs*.

Simulated RXD-Input Character Stream

You can enter a character stream that is simulated entering the RXD port upon simulated system reset. The UART core's MegaWizard interface accepts an arbitrary character string, which is later incorporated into the UART simulation model. After reset in reset, the string is input into the RXD port character-by-character as the core is able to accept new data.

Prepare Interactive Windows

At system generation time, the UART core generator can create ModelSim macros that facilitate interaction with the UART model during simulation. The following options are available:

Create ModelSim Alias to Open Streaming Output Window
A ModelSim macro is created to open a window that displays all output from the TXD port.

Create ModelSim Alias to Open Interactive Stimulus Window
A ModelSim macro is created to open a window that accepts stimulus for the RXD port. The window sends any characters typed in the window to the RXD port.

Simulated Transmitter Baud Rate

RS-232 transmission rates are often slower than any other process in the system, and it is seldom useful to simulate the functional model at the true baud rate. For example, at 115,200 bps, it typically takes thousands of clock cycles to transfer a single character. The UART simulation model has the ability to run with a constant clock divisor of 2. This allows the simulated UART to transfer bits at half the system clock speed, or roughly one character per 20 clock cycles. You can choose one of the following options for the simulated transmitter baud rate:

- **accelerated (use divisor = 2)**—`TXD` emits one bit per 2 clock cycles in simulation.
- **actual (use true baud divisor)**—`TXD` transmits at the actual baud rate, as determined by the `divisor` register.

Hardware Simulation Considerations

The simulation features were created for easy simulation of Nios, Nios II or Excalibur™ processor systems when using the ModelSim simulator. The documentation for each processor documents the suggested usage of these features. Other usages may be possible, but will require additional user effort to create a custom simulation process.

The simulation model is implemented in the UART core's top-level HDL file; the synthesizable HDL and the simulation HDL are implemented in the same file. The simulation features are implemented using `translate on` and `translate off` synthesis directives that make certain sections of HDL code visible only to the synthesis tool.

Do not edit the simulation directives if you are using Altera's recommended simulation procedures. If you do change the simulation directives for your custom simulation flow, be aware that SOPC Builder overwrites existing files during system generation. Take precaution so that your changes are not overwritten.



For details about simulating the UART core in Nios II processor systems, refer to *AN 351: Simulating Nios II Processor Designs*. For details about simulating the UART core in Nios embedded processor systems, refer to *AN 189: Simulating Nios Embedded Processor Designs*.

Software Programming Model

The following sections describe the software programming model for the UART core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides hardware abstraction layer (HAL) system library drivers that enable you to access the UART core using the ANSI C standard library functions, such as `printf()` and `getchar()`.

HAL System Library Support

The Altera-provided driver implements a HAL character-mode device driver that integrates into the HAL system library for Nios II systems. HAL users should access the UART via the familiar HAL API and the ANSI C standard library, rather than accessing the UART registers. `ioctl()` requests are defined that allow HAL users to control the hardware-dependent aspects of the UART.



If your program uses the HAL device driver to access the UART hardware, accessing the device registers directly will interfere with the correct behavior of the driver.

For Nios II processor users, the HAL system library API provides complete access to the UART core's features. Nios II programs treat the UART core as a character mode device, and send and receive data using the ANSI C standard library functions.

The driver supports the CTS/RTS control signals when they are enabled in SOPC Builder. Refer to "Driver Options: Fast Versus Small Implementations" on page 9–11.

The following code demonstrates the simplest possible usage, printing a message to stdout using `printf()`. In this example, the SOPC Builder system contains a UART core, and the HAL system library has been configured to use this device for stdout.

Example 9–1. Example: Printing Characters to a UART Core as stdout

```
#include <stdio.h>
int main ()
{
  printf("Hello world.\n");
  return 0;
}
```

The following code demonstrates reading characters from and sending messages to a UART device using the C standard library. In this example, the SOPC Builder system contains a UART core named `uart1` that is not necessarily configured as the stdout device. In this case, the program treats the device like any other node in the HAL file system.

Example 9–2. Example: Sending and Receiving Characters

```
/* A simple program that recognizes the characters 't' and 'v' */
#include <stdio.h>
#include <string.h>
int main ()
{
  char* msg = "Detected the character 't'.\n";
  FILE* fp;
  char prompt = 0;

  fp = fopen ("/dev/uart1", "r+"); //Open file for reading and writing
  if (fp)
  {
    while (prompt != 'v')
    {  // Loop until we receive a 'v'.
      prompt = getc(fp);  // Get a character from the UART.
      if (prompt == 't')
      {  // Print a message if character is 't'.
        fwrite (msg, strlen (msg), 1, fp);
      }
    }

    fprintf(fp, "Closing the UART file.\n");
    fclose (fp);
  }

  return 0;
}
```



For more information about the HAL system library, refer to the *Nios II Software Developer's Handbook*.

Driver Options: Fast Versus Small Implementations

To accommodate the requirements of different types of systems, the UART driver provides two variants: a fast version and a small version. The fast version is the default. Both fast and small drivers fully support the C standard library functions and the HAL API.

The fast driver is an interrupt-driven implementation, which allows the processor to perform other tasks when the device is not ready to send or receive data. Because the UART data rate is slow compared to the processor, the fast driver can provide a large performance benefit for systems that could be performing other tasks in the interim.

The small driver is a polled implementation that waits for the UART hardware before sending and receiving each character. There are two ways to enable the small footprint driver:

- Enable the small footprint setting for the HAL system library project. This option affects device drivers for all devices in the system as well.
- Specify the preprocessor option `-DALTERA_AVALON_UART_SMALL`. You can use this option if you want the small, polled implementation of the UART driver, but do not want to affect the drivers for other devices.



Refer to the help system in the Nios II IDE for details about how to set HAL properties and preprocessor options.

If the CTS/RTS flow control signals are enabled in hardware, the fast driver automatically uses them. The small driver always ignores them.

ioctl() Operations

The UART driver supports the `ioctl()` function to allow HAL-based programs to request device-specific operations. Table 9–2 defines operation requests that the UART driver supports.

Table 9–2. UART ioctl() Operations

Request	Meaning
`TIOCEXCL`	Locks the device for exclusive access. Further calls to `open()` for this device will fail until either this file descriptor is closed, or the lock is released using the TIOCNXCL `ioctl` request. For this request to succeed there can be no other existing file descriptors for this device. The `ioctl` "arg" parameter is ignored.
`TIOCNXCL`	Releases a previous exclusive access lock. The `ioctl` "arg" parameter is ignored.

Additional operation requests are also optionally available for the fast driver only, as shown in Table 9–3. To enable these operations in your program, you must set the preprocessor option `-DALTERA_AVALON_UART_USE_IOCTL`.

Table 9–3. Optional UART ioctl() Operations for the Fast Driver Only

Request	Meaning
`TIOCMGET`	Returns the current configuration of the device by filling in the contents of the input termios *(1)* structure. A pointer to this structure is supplied as the value of the `ioctl` "opt" parameter.
`TIOCMSET`	Sets the configuration of the device according to the values contained in the input termios structure *(1)*. A pointer to this structure is supplied as the value of the `ioctl` "arg" parameter.

Note to Table 9–3:

(1) The termios structure is defined by the Newlib C standard library. You can find the definition in the file <*Nios II EDS install path*>**/components/altera_hal/HAL/inc/sys/termios.h**.



Refer to the *Nios II Software Developer's Handbook* for details about the `ioctl()` function.

Limitations

The HAL driver for the UART core does not support the endofpacket register. Refer to "Register Map" for details.

Software Files

The UART core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_uart_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- **altera_avalon_uart.h**, **altera_avalon_uart.c**—These files implement the UART core device driver for the HAL system library.

Legacy SDK Routines

The UART core is also supported by the legacy SDK routines for the first-generation Nios processor.

For details about these routines, refer to the UART documentation that accompanied the first-generation Nios processor. For details about upgrading programs based on the legacy SDK to the HAL system library API, refer to *AN 350: Upgrading Nios Processor Systems to the Nios II Processor*.

Register Map

Programmers using the HAL API or the legacy SDK for the first-generation Nios processor never access the UART core directly via its registers. In general, the register map is only useful to programmers writing a device driver for the core.

CAUTION

The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver and the HAL driver is active for the same device, your driver will conflict and fail to operate.

Table 9–4 shows the register map for the UART core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 9–4. UART Core Register Map

Offset	Register Name	R/W	Description/Register Bits														
			15 . . .13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	rxdata	RO	(1)					(2)	(2)	Receive Data						
1	txdata	WO	(1)					(2)	(2)	Transmit Data						
2	status (3)	RW	(1)	eop	cts	dcts	(1)	e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	control	RW	(1)	ieop	rts	idcts	trbk	ie	irrdy	itrdy	itmt	itoe	iroe	ibrk	ife	ipe
4	divisor (4)	RW	Baud Rate Divisor													
5	endof-packet (4)	RW	(1)					(2)	(2)	End-of-Packet Value						

Notes to Table 9–4:

(1) These bits are reserved. Reading returns an undefined value. Write zero.

(2) These bits may or may not exist, depending on the **Data Width** hardware option. If they do not exist, they read zero, and writing has no effect.

(3) Writing zero to the status register clears the dcts, e, toe, roe, brk, fe, and PE bits.

(4) This register may or may not exist, depending on hardware configuration options. If it does not exist, reading returns an undefined value and writing has no effect.

Some registers and bits are optional. These registers and bits exists in hardware only if it was enabled at system generation time. Optional registers and bits are noted in the following sections.

rxdata Register

The rxdata register holds data received via the RXD input. When a new character is fully received via the RXD input, it is transferred into the rxdata register, and the status register's rrdy bit is set to 1. The status register's rrdy bit is set to 0 when the rxdata register is read. If a character is transferred into the rxdata register while the rrdy bit is already set (in other words, the previous character was not retrieved), a receiver-overrun error occurs and the status register's roe bit is set to 1. New characters are always transferred into the rxdata register, regardless of whether the previous character was read. Writing data to the rxdata register has no effect.

txdata Register

Avalon-MM master peripherals write characters to be transmitted into the txdata register. Characters should not be written to txdata until the transmitter is ready for a new character, as indicated by the TRDY bit in the status register. The TRDY bit is set to 0 when a character is written into the txdata register. The TRDY bit is set to 1 when the character is transferred from the txdata register into the transmitter shift register. If a character is written to the txdata register when TRDY is 0, the result is undefined. Reading the txdata register returns an undefined value.

For example, assume the transmitter logic is idle and an Avalon-MM master peripheral writes a first character into the txdata register. The TRDY bit is set to 0, then set to 1 when the character is transferred into the transmitter shift register. The master can then write a second character into the txdata register, and the TRDY bit is set to 0 again. However, this time the shift register is still busy shifting out the first character to the TXD output. The TRDY bit is not set to 1 until the first character is fully shifted out and the second character is automatically transferred into the transmitter shift register.

status Register

The status register consists of individual bits that indicate particular conditions inside the UART core. Each status bit is associated with a corresponding interrupt-enable bit in the control register. The status register can be read at any time. Reading does not change the value of any of the bits. Writing zero to the status register clears the DCTS, E, TOE, ROE, BRK, FE, and PE bits.

The status register bits are shown in Table 9–5.

Table 9–5. status Register Bits (Part 1 of 3)

Bit	Bit Name	Read/ Write/ Clear	Description
0 *(1)*	PE	RC	Parity error. A parity error occurs when the received parity bit has an unexpected (incorrect) logic level. The PE bit is set to 1 when the core receives a character with an incorrect parity bit. The PE bit stays set to 1 until it is explicitly cleared by a write to the status register. When the PE bit is set, reading from the rxdata register produces an undefined value. If the **Parity** hardware option is not enabled, no parity checking is performed and the PE bit always reads 0. Refer to "Data Bits, Stop Bits, Parity" on page 9–6.
1	FE	RC	Framing error. A framing error occurs when the receiver fails to detect a correct stop bit. The FE bit is set to 1 when the core receives a character with an incorrect stop bit. The FE bit stays set to 1 until it is explicitly cleared by a write to the status register. When the FE bit is set, reading from the rxdata register produces an undefined value.
2	BRK	RC	Break detect. The receiver logic detects a break when the RXD pin is held low (logic 0) continuously for longer than a full-character time (data bits, plus start, stop, and parity bits). When a break is detected, the BRK bit is set to 1. The BRK bit stays set to 1 until it is explicitly cleared by a write to the status register.
3	ROE	RC	Receive overrun error. A receive-overrun error occurs when a newly received character is transferred into the rxdata holding register before the previous character is read (in other words, while the RRDY bit is 1). In this case, the ROE bit is set to 1, and the previous contents of rxdata are overwritten with the new character. The ROE bit stays set to 1 until it is explicitly cleared by a write to the status register.
4	TOE	RC	Transmit overrun error. A transmit-overrun error occurs when a new character is written to the txdata holding register before the previous character is transferred into the shift register (in other words, while the TRDY bit is 0). In this case the TOE bit is set to 1. The TOE bit stays set to 1 until it is explicitly cleared by a write to the status register.
5	TMT	R	Transmit empty. The TMT bit indicates the transmitter shift register's current state. When the shift register is in the process of shifting a character out the TXD pin, TMT is set to 0. When the shift register is idle (in other words, a character is not being transmitted) the TMT bit is 1. An Avalon-MM master peripheral can determine if a transmission is completed (and received at the other end of a serial link) by checking the TMT bit.

Table 9–5. status Register Bits (Part 2 of 3)

Bit	Bit Name	Read/ Write/ Clear	Description
6	TRDY	R	Transmit ready. The TRDY bit indicates the `txdata` holding register's current state. When the `txdata` register is empty, it is ready for a new character, and TRDY is 1. When the `txdata` register is full, TRDY is 0. An Avalon-MM master peripheral must wait for TRDY to be 1 before writing new data to `txdata`.
7	RRDY	R	Receive character ready. The RRDY bit indicates the `rxdata` holding register's current state. When the `rxdata` register is empty, it is not ready to be read and rrdy is 0. When a newly received value is transferred into the `rxdata` register, RRDY is set to 1. Reading the `rxdata` register clears the RRDY bit to 0. An Avalon-MM master peripheral must wait for RRDY to equal 1 before reading the `rxdata` register.
8	E	RC	Exception. The E bit indicates that an exception condition occurred. The E bit is a logical-OR of the TOE, ROE, BRK, FE, and PE bits. The e bit and its corresponding interrupt-enable bit (IE) bit in the `control` register provide a convenient method to enable/disable IRQs for all error conditions. The E bit is set to 0 by a write operation to the status register.
10 *(1)*	DCTS	RC	Change in clear to send (CTS) signal. The DCTS bit is set to 1 whenever a logic-level transition is detected on the `CTS_N` input port (sampled synchronously to the Avalon-MM clock). This bit is set by both falling and rising transitions on `CTS_N`. The DCTS bit stays set to 1 until it is explicitly cleared by a write to the `status` register. If the **Flow Control** hardware option is not enabled, the DCTS bit always reads 0. Refer to "Flow Control" on page 9–7.
11 *(1)*	CTS	R	Clear-to-send (CTS) signal. The CTS bit reflects the `CTS_N` input's instantaneous state (sampled synchronously to the Avalon-MM clock). Because the `CTS_N` input is logic negative, the CTS bit is 1 when a 0 logic-level is applied to the `CTS_N` input. The `CTS_N` input has no effect on the transmit or receive processes. The only visible effect of the `CTS_N` input is the state of the CTS and DCTS bits, and an IRQ that can be generated when the control register's idcts bit is enabled. If the **Flow Control** hardware option is not enabled, the CTS bit always reads 0. Refer to "Flow Control" on page 9–7.

Table 9–5. status Register Bits (Part 3 of 3)

Bit	Bit Name	Read/ Write/ Clear	Description
12 *(1)*	EOP	R	End of packet encountered. The EOP bit is set to 1 by one of the following events: • An EOP character is written to `txdata` • An EOP character is read from `rxdata` The EOP character is determined by the contents of the `endofpacket` register. The EOP bit stays set to 1 until it is explicitly cleared by a `write` to the `status` register. If the **Include End-of-Packet Register** hardware option is not enabled, the EOP bit always reads 0. Refer to "Avalon-MM Transfers with Flow Control (DMA)" on page 9–7.

Note to Table 9–5:
(1) This bit is optional and may not exist in hardware.

control Register

The `control` register consists of individual bits, each controlling an aspect of the UART core's operation. The value in the `control` register can be read at any time.

Each bit in the `control` register enables an IRQ for a corresponding bit in the `status` register. When both a status bit and its corresponding interrupt-enable bit are 1, the core generates an IRQ. For example, the PE bit is bit 0 of the `status` register, and the ipe bit is bit 0 of the `control` register. An interrupt request is generated when both PE and ipe equal 1.

The control register bits are shown in Table 9–6.

Table 9–6. control Register Bits (Part 1 of 2)

Bit	Bit Name	Read/ Write	Description
0	IPE	RW	Enable interrupt for a parity error.
1	IFE	RW	Enable interrupt for a framing error.
2	IBRK	RW	Enable interrupt for a break detect.
3	IROE	RW	Enable interrupt for a receiver overrun error.
4	ITOE	RW	Enable interrupt for a transmitter overrun error.
5	ITMT	RW	Enable interrupt for a transmitter shift register empty.

Table 9–6. control Register Bits (Part 2 of 2)

Bit	Bit Name	Read/ Write	Description
6	ITRDY	RW	Enable interrupt for a transmission ready.
7	IRRDY	RW	Enable interrupt for a read ready.
8	IE	RW	Enable interrupt for an exception.
9	TRBK	RW	Transmit break. The TRBK bit allows an Avalon-MM master peripheral to transmit a break character over the TXD output. The TXD signal is forced to 0 when the TRBK bit is set to 1. The TRBK bit overrides any logic level that the transmitter logic would otherwise drive on the TXD output. The TRBK bit interferes with any transmission in process. The Avalon-MM master peripheral must set the TRBK bit back to 0 after an appropriate break period elapses.
10	IDCTS	RW	Enable interrupt for a change in CTS signal.
11 *(1)*	RTS	RW	Request to send (RTS) signal. The RTS bit directly feeds the RTS_N output. An Avalon-MM master peripheral can write the RTS bit at any time. The value of the RTS bit only affects the RTS_N output; it has no effect on the transmitter or receiver logic. Because the RTS_N output is logic negative, when the RTS bit is 1, a low logic-level (0) is driven on the RTS_N output. If the **Flow Control** hardware option is not enabled, the RTS bit always reads 0, and writing has no effect. Refer to "Flow Control" on page 9–7.
12	IEOP	RW	Enable interrupt for end-of-packet condition.

Note to Table 9–6:
(1) This bit is optional and may not exist in hardware.

divisor Register (Optional)

The value in the divisor register is used to generate the baud rate clock. The effective baud rate is determined by the formula shown in Equation 3:

$$\text{baud rate} = \frac{\text{clock frequency}}{(\text{divisor} + 1)} \quad (3)$$

The divisor register is an optional hardware feature. If the **Baud Rate Can Be Changed By Software** hardware option is not enabled, then the divisor register does not exist. In this case, writing divisor has no effect, and reading divisor returns an undefined value. For more information, refer to "Baud Rate Options" on page 9–5.

endofpacket Register (Optional)

The value in the `endofpacket` register determines the end-of-packet character for variable-length DMA transactions. After reset, the default value is zero, which is the ASCII null character (\0). For more information, refer to Table 9–5 on page 9–16 for the description for the eop bit.

The `endofpacket` register is an optional hardware feature. If the **Include end-of-packet register** hardware option is not enabled, then the `endofpacket` register does not exist. In this case, writing `endofpacket` has no effect, and reading returns an undefined value.

Interrupt Behavior

The UART core outputs a single IRQ signal to the Avalon-MM interface, which can connect to any master peripheral in the system, such as a Nios II processor. The master peripheral must read the `status` register to determine the cause of the interrupt.

Every interrupt condition has an associated bit in the `status` register and an interrupt-enable bit in the `control` register. When any of the interrupt conditions occur, the associated `status` bit is set to 1 and remains set until it is explicitly acknowledged. The IRQ output is asserted when any of the status bits are set while the corresponding interrupt-enable bit is 1. A master peripheral can acknowledge the IRQ by clearing the status register.

At reset, all interrupt-enable bits are set to 0; therefore, the core cannot assert an IRQ until a master peripheral sets one or more of the interrupt-enable bits to 1.

All possible interrupt conditions are listed with their associated status and control (interrupt-enable) bits in Table 6–5 on page 6–16 and Table 6–6 on page 6–18. Details of each interrupt condition are provided in the `status` bit descriptions.

Referenced Documents

This chapter references the following documents:

- *Nios II Software Developer's Handbook*
- *Timer Core chapter in volume 5 of the Quartus II Handbook*
- *Avalon Memory-Mapped Interface Specification*
- *AN 351: Simulating Nios II Processor Designs*
- *AN 189: Simulating Nios Embedded Processor Designs*

Document Revision History

Table 9–7 shows the revision history for this chapter.

Table 9–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.	• Chapter 9 was formerly Chapter 8. • Added two sentences to clarify use of flow control. Host PC must also be configured for flow control.	—
May 2007 v7.1.0	• Chapter 8 was formerly chapter 6. • Added table of contents to Overview section. • Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies. Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface." • Corrected definition of even and odd parity in section "*Data Bits, Stop Bits, Parity*" on page 8–6.	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology. Other changes to the document serve only to clarify existing behavior.
May 2006 v6.0.0	No change from previous release.	—
December 2005 v5.1.1	Changed Avalon "streaming" terminology to "flow control" based on a change to the *Avalon Interface Specification.*	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

10. SPI Core

NII51011-7.2.0

Core Overview

SPI is an industry-standard serial protocol commonly used in embedded systems to connect microprocessors to a variety of off-chip sensor, conversion, memory, and control devices. The SPI core with Avalon® interface implements the SPI protocol and provides an Avalon Memory-Mapped (Avalon-MM) interface on the back end.

The SPI core can implement either the master or slave protocol. When configured as a master, the SPI core can control up to 16 independent SPI slaves. The width of the receive and transmit registers are configurable between 1 and 16 bits. Longer transfer lengths (for example, 24-bit transfers) can be supported with software routines. The SPI core provides an interrupt output that can flag an interrupt whenever a transfer completes.

The SPI core is SOPC Builder ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Instantiating the SPI Core in SOPC Builder" on page 10–7
- "Device and Tools Support" on page 10–10
- "Software Programming Model" on page 10–10

Functional Description

The SPI core communicates using two data lines, a control line, and a synchronization clock:

- **Master Out Slave In** (`mosi`)—Output data from the master to the inputs of the slaves
- **Master In Slave Out** (`miso`)—Output data from a slave to the input of the master
- **Serial Clock** (`sclk`)—Clock driven by the master to slaves, used to synchronize the data bits
- **Slave Select** (`ss_n`)— Select signal (active low) driven by the master to individual slaves, used to select the target slave

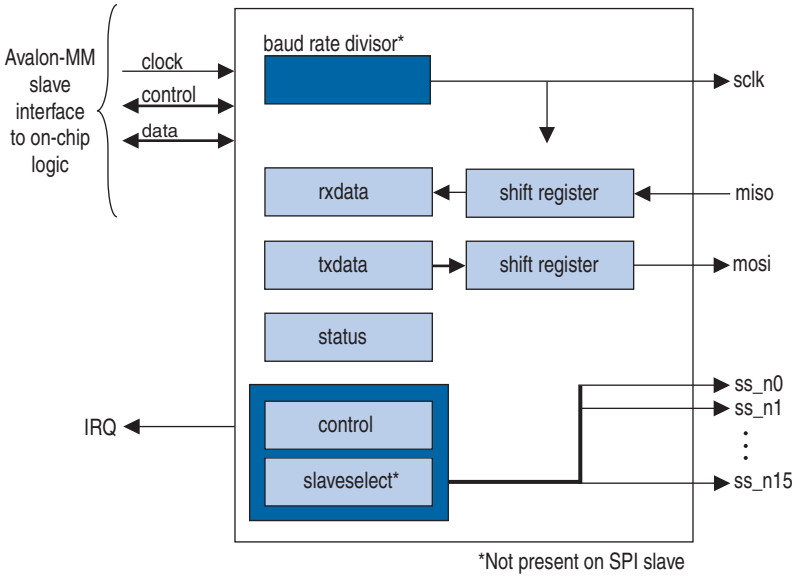
The SPI core has the following user-visible features:

- A memory-mapped register space comprised of five registers: `rxdata`, `txdata`, `status`, `control`, and `slaveselect`
- Four SPI interface ports: `sclk`, `ss_n`, `mosi`, and `miso`

The registers provide an interface to the SPI core and are visible via the Avalon-MM slave port. The `sclk`, `ss_n`, `mosi`, and `miso` ports provide the hardware interface to other SPI devices. The behavior of `sclk`, `ss_n`, `mosi`, and `miso` depends on whether the SPI core is configured as a master or slave.

Figure 10–1 shows a block diagram of the SPI core in master mode.

Figure 10–1. SPI Core Block Diagram

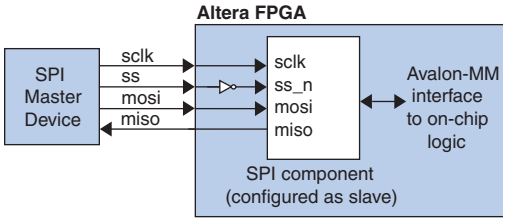


The SPI core logic is synchronous to the clock input provided by the Avalon-MM interface. When configured as a master, the core divides the Avalon-MM clock to generate the SCLK output. When configured as a slave, the core's receive logic is synchronized to SCLK input. The core's Avalon-MM interface is capable of Avalon-MM transfers with flow control. The SPI core can be used in conjunction with a DMA controller with flow control to automate continuous data transfers between, for example, the SPI core and memory. See the *Timer Core* chapter for details.

Example Configurations

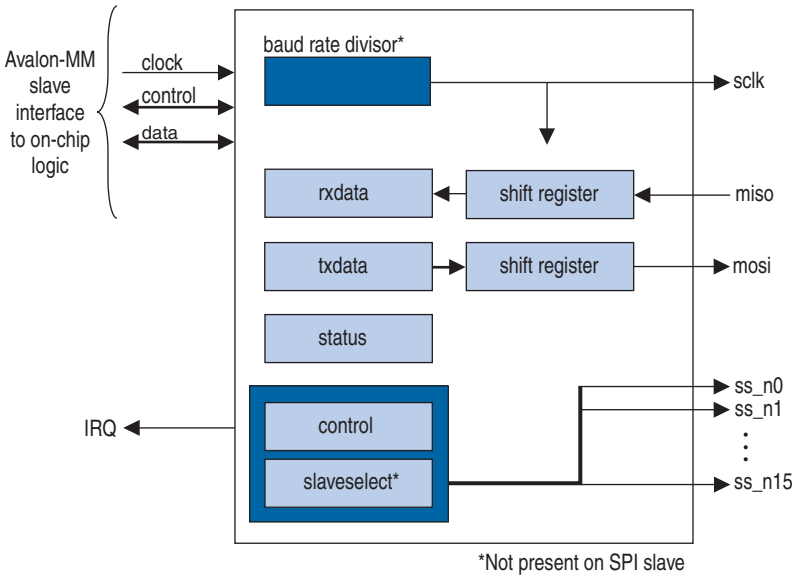
Two possible configurations are shown below. In Figure 10–2, the SPI core provides a slave interface to an off-chip SPI master.

Figure 10–2. SPI Core Configured as a Slave



In Figure 10–3 the SPI core provides a master interface driving multiple off-chip slave devices. Each slave device in Figure 10–3 must tristate its `miso` output whenever its select signal is not asserted.

Figure 10–3. SPI Core Configured as a Master



The `ss_n` signal is active-low. However, any signal can be inverted inside the FPGA, allowing the slave-select signals to be either active high or active low.

Transmitter Logic

The SPI core transmitter logic consists of a transmit holding register (`txdata`) and transmit shift register, each *n* bits wide. The register width *n* is specified at system generation time, and can be any integer value from 1 to 16. After a master peripheral writes a value to the `txdata` register, the value is copied to the shift register and then transmitted when the next operation starts.

The shift register and the `txdata` register provide double buffering during data transmission. A new value can be written into the `txdata` register while the previous data is being shifted out of the shift register. The transmitter logic automatically transfers the `txdata` register to the shift register whenever a serial shift operation is not currently in process.

In master mode, the transmit shift register directly feeds the `mosi` output. In slave mode, the transmit shift register directly feeds the `miso` output. Data shifts out least-significant bit (LSB) first or most-significant bit (MSB) first, depending on the configuration of the SPI core.

Receiver Logic

The SPI core receive logic consists of a receive holding register (`rxdata`) and receive shift register, each *n* bits wide. The register width *n* is specified at system generation time, and can be any integer value from 1 to 16. A master peripheral reads received data from the `rxdata` register after the shift register has captured a full *n*-bit value of data.

The shift register and the `rxdata` register provide double buffering during data receiving. The `rxdata` register can hold a previously received data value while subsequent new data is shifting into the shift register. The receiver logic automatically transfers the shift register content to the `rxdata` register when a serial shift operation completes.

In master mode, the shift register is fed directly by the `miso` input. In slave mode, the shift register is fed directly by the `mosi` input. The receiver logic expects input data to arrive least-significant bit (LSB) first or most-significant bit (MSB) first, depending on the configuration of the SPI core.

Master and Slave Modes

At system generation time, the designer configures the SPI core in either master mode or slave mode. The mode cannot be switched at runtime.

Master Mode Operation

In master mode, the SPI ports behave as shown in Table 10–1.

Table 10–1. Master Mode Port Configurations

Name	Direction	Description
mosi	output	Data output to slave(s)
miso	input	Data input from slave(s)
sclk	output	Synchronization clock to all slaves
ss_n*M*	output	Slave select signal to slave *M*, where M is a number between 0 and 15.

Only an SPI master can initiate an operation between master and slave. In master mode, an intelligent host (for example, a microprocessor) configures the SPI core using the control and slaveselect registers, and then writes data to the txdata buffer to initiate a transaction. A master peripheral can monitor the status of the transaction by reading the status register. A master peripheral can enable interrupts to notify the host whenever new data is received (that is, a transfer has completed), or whenever the transmit buffer is ready for new data.

The SPI protocol is full duplex, so every transaction both sends and receives data at the same time. The master transmits a new data bit on the mosi output and the slave drives a new data bit on the miso input for each active edge of sclk. The SPI core divides the Avalon-MM system clock using a clock divider to generate the sclk signal.

When the SPI core is configured to interface with multiple slaves, the core has one ss_n signal for each slave, up to a maximum of sixteen slaves. During a transfer, the master asserts ss_n to each slave specified in the slaveselect register. Note that there can be no more than one slave transmitting data during any particular transfer, or else there will be a conflict on the miso input. The number of slave devices is specified at system generation time.

Slave Mode Operation

In slave mode, the SPI ports behave as shown in Table 10–2.

Table 10–2. Slave Mode Port Configurations

Name	Direction	Description
mosi	input	Data input from the master
miso	output	Data output to the master
sclk	input	Synchronization clock
ss_n	input	Select signal

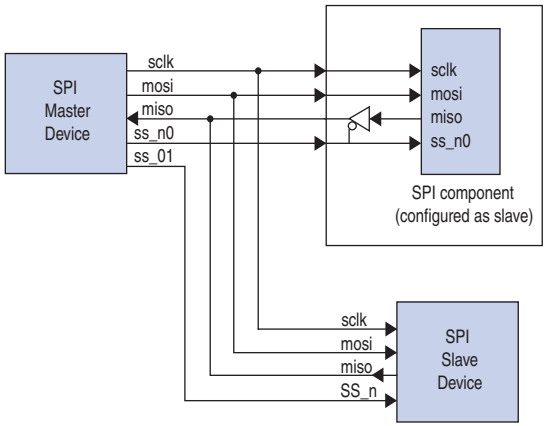
In slave mode, the SPI core simply waits for the master to initiate transactions. Before a transaction begins, the slave logic is continuously polling the ss_n input. When the master asserts ss_n (drives it low), the slave logic immediately begins sending the transmit shift register contents to the miso output. The slave logic also captures data on the mosi input, and fills the receive shift register simultaneously. Thus, a read and write transaction are carried out simultaneously.

An intelligent host (for example, a microprocessor) writes data to the txdata registers, so that it will be transmitted the next time the master initiates an operation. A master peripheral reads received data from the rxdata register. A master peripheral can enable interrupts to notify the host whenever new data is received, or whenever the transmit buffer is ready for new data.

Multi-Slave Environments

When ss_n is not asserted, typical SPI cores set their miso output pins to high impedance. The Altera®-provided SPI slave core drives an undefined high or low value on its miso output when not selected. Special consideration is necessary to avoid signal contention on the miso output, if the SPI core in slave mode will be connected to an off-chip SPI master device with multiple slaves. In this case, the ss_n input should be used to control a tristate buffer on the miso signal. Figure 10–4 shows an example of the SPI core in slave mode in an environment with two slaves.

Figure 10–4. SPI Core in a Multi-Slave Environment



Avalon-MM Interface

The SPI core's Avalon-MM interface consists of a single Avalon-MM slave port. In addition to fundamental slave read and write transfers, the SPI core supports Avalon-MM read and write transfers with flow control.

Instantiating the SPI Core in SOPC Builder

Designers use the MegaWizard® interface for the SPI core in SOPC Builder to configure the hardware feature set. The following sections describe the available options.

Master/Slave Settings

The designer can select either master mode or slave mode to determine the role of the SPI core. When master mode is selected, the following options are available:

- "Generate Select Signals"
- "SPI Clock (sclk) Rate" on page 10–8
- "Specify Delay" on page 10–8

Generate Select Signals

This setting specifies how many slaves the SPI master will connect to. The acceptable range is 1 to 16. The SPI master core presents a unique `ss_n` signal for each slave.

SPI Clock (sclk) Rate

This setting determines the rate of the `sclk` signal that synchronizes data between master and slaves. The target clock rate can be specified in units of Hz, kHz or MHz. The SPI master core uses the Avalon-MM system clock and a clock divisor to generate `sclk`.

The actual frequency of `sclk` may not exactly match the desired target clock rate. The achievable clock values are:

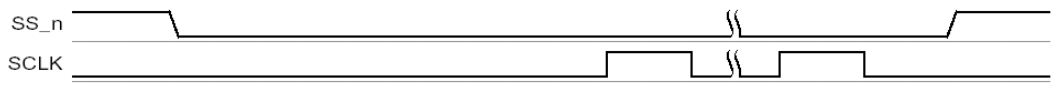
<Avalon-MM system clock frequency> / [2, 4, 6, 8, ...]

The actual frequency achieved will not be greater than the specified target value. For example, if the system clock frequency is 50 MHz and the target value is 25 MHz, then the clock divisor is 2 and the actual `sclk` frequency achieves exactly 25 MHz. However, if the target frequency is 24 MHz, then the clock divisor is 4 and the actual `sclk` frequency becomes 12.5 MHz.

Specify Delay

Turning on this option causes the SPI master to add a time delay between asserting the `ss_n` signal and shifting the first bit of data. This delay is required by certain SPI slave devices. If the delay option is on, the designer must also specify the delay time in units of ns, us or ms. An example is shown in Figure 10–5.

Figure 10–5. Time Delay Between Asserting ss_n and Toggling sclk



The delay generation logic uses a granularity of half the period of `sclk`. The actual delay achieved is the desired target delay rounded up to the nearest multiple of half the `sclk` period, as shown in the following equations:

(1) $$p = \frac{1}{2}(\text{period of sclk})$$

(2) $$\text{actual delay} = \text{ceiling}\left(\frac{\text{<desired delay>}}{p}\right) \times p$$

Data Register Settings

The data register settings affect the size and behavior of the data registers in the SPI core. There are two data register settings:

- *Width*—This setting specifies the width of `rxdata`, `txdata`, and the receive and transmit shift registers. Acceptable values are from 1 to 16.
- *Shift direction*—This setting determines the direction that data shifts (MSB first or LSB first) into and out of the shift registers.

Timing Settings

The timing settings affect the timing relationship between the `ss_n`, `sclk`, `mosi` and `miso` signals. In this discussion the `mosi` and `miso` signals are referred to generically as "data". There are two timing settings:

- *Clock polarity*—This setting can be 0 or 1. When clock polarity is set to 0, the idle state for `sclk` is low. When clock polarity is set to 1, the idle state for `sclk` is high.
- *Clock phase*—This setting can be 0 or 1. When clock phase is 0, data is latched on the leading edge of `sclk`, and data changes on trailing edge. When clock phase is 1, data is latched on the trailing edge of `sclk`, and data changes on the leading edge.

Figures 10–6 through 10–9 demonstrate the behavior of signals in all possible cases of clock polarity and clock phase.

Figure 10–6. Clock Polarity = 0, Clock Phase = 0

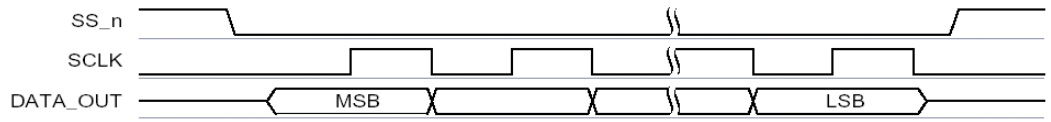


Figure 10–7. Clock Polarity = 0, Clock Phase = 1

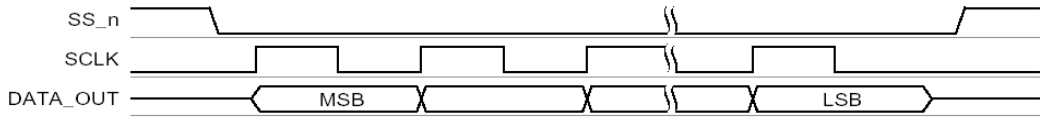


Figure 10–8. Clock Polarity = 1, Clock Phase = 0

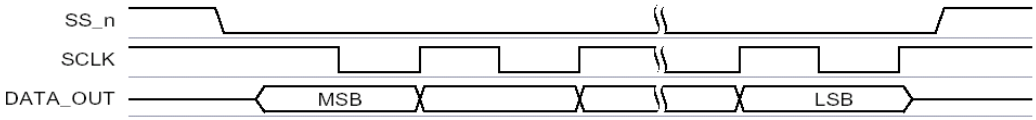
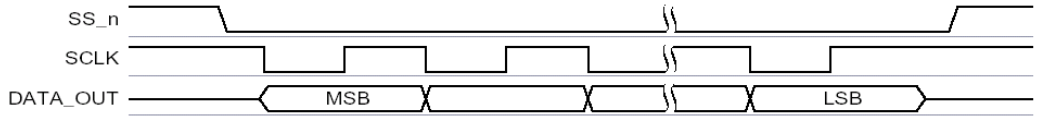


Figure 10–9. Clock Polarity = 1, Clock Phase = 1



Device and Tools Support

The SPI core can target all Altera FPGAs.

Software Programming Model

The following sections describe the software programming model for the SPI core, including the register map and software constructs used to access the hardware. For Nios® II processor users, Altera provides the HAL system library header file that defines the SPI core registers. The SPI core does not match the generic device model categories supported by the HAL, so it cannot be accessed via the HAL API or the ANSI C standard library. Altera provides a routine to access the SPI hardware that is specific to the SPI core.

Hardware Access Routines

Altera provides one access routine, `alt_avalon_spi_command()`, that provides general-purpose access to an SPI core configured as a master.

alt_avalon_spi_command()

Prototype:

```
int alt_avalon_spi_command(alt_u32 base, alt_u32 slave,
                           alt_u32 write_length,
                           const alt_u8* wdata,
                           alt_u32 read_length,
                           alt_u8* read_data,
                           alt_u32 flags)
```

Thread-safe: No.

Available from ISR: No.

Include: **<altera_avalon_spi.h>**

Description: `alt_avalon_spi_command()` is used to perform a control sequence on the SPI bus. This routine is designed for SPI masters of 8-bit data width or less. Currently, it does not support SPI hardware with data-width greater than 8 bits. A single call to this function writes a data buffer of arbitrary length out the MOSI port, and then reads back an arbitrary amount of data from the MISO port. The function performs the following actions:

(1) Asserts the slave select output for the specified slave. The first slave select output is numbered 0, the next is 1, etc.
(2) Transmits `write_length` bytes of data from `wdata` through the SPI interface, discarding the incoming data on MISO.
(3) Reads `read_length` bytes of data, storing the data into the buffer pointed to by `read_data`. MOSI is set to zero during the read transaction.
(4) De-asserts the slave select output, unless the flags field contains the value `ALT_AVALON_SPI_COMMAND_MERGE`. If you want to transmit from scattered buffers then you can call the function multiple times, specifying the merge flag on all the accesses except the last.

This function is not thread safe. If you want to access the SPI bus from more than one thread, then you should use a semaphore or mutex to ensure that only one thread is executing within this function at any time.

Returns: The number of bytes stored in the `read_data` buffer.

Software Files

The SPI core is accompanied by the following software files. These files provide a low-level interface to the hardware.

- **altera_avalon_spi.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_spi.c**—This file implements low-level routines to access the hardware.

Legacy SDK Routines

The SPI core is also supported by the legacy SDK routines for the first-generation Nios processor. For details about these routines, refer to the SPI documentation that accompanied the first-generation Nios processor.



For details about upgrading programs based on the legacy SDK to the HAL system library API, refer to *AN 350: Upgrading Nios Processor Systems to the Nios II Processor*.

Register Map

An Avalon-MM master peripheral controls and communicates with the SPI core via the six 16-bit registers, shown in Table 10–3. The table assumes an *n*-bit data width for `rxdata` and `txdata`.

Table 10–3. Register Map for SPI Master Device

Internal Address	Register Name	15…11	10	9	8	7	6	5	4	3	2	1	0
0	`rxdata` *(1)*	RXDATA (n-1..0)											
1	`txdata` *(1)*	TXDATA (n-1..0)											
2	`status` *(2)*				E	RRDY	TRDY	TMT	TOE	ROE			
3	`control`		sso *(3)*		IE	IRRDY	ITRDY		ITOE	IROE			
4	Reserved												
5	`slaveselect` *(3)*	Slave Select Mask											

Notes to Table 10–3:
(1) Bits 15 to *n* are undefined when *n* is less than 16.
(2) A write operation to the `status` register clears the `roe`, `toe` and `e` bits.
(3) Present only in master mode.

Reading undefined bits returns an undefined value. Writing to undefined bits has no effect.

rxdata Register

A master peripheral reads received data from the rxdata register. When the receive shift register receives a full *n* bits of data, the status register's rrdy bit is set to 1 and the data is transferred into the rxdata register. Reading the rxdata register clears the rrdy bit. Writing to the rxdata register has no effect.

New data is always transferred into the rxdata register, whether or not the previous data was retrieved. If rrdy is 1 when data is transferred into the rxdata register (that is, the previous data was not retrieved), a receive-overrun error occurs and the status register's roe bit is set to 1. In this case, the contents of rxdata are undefined.

txdata Register

A master peripheral writes data to be transmitted into the txdata register. When the status register's trdy bit is 1, it indicates that the txdata register is ready for new data. The trdy bit is set to 0 whenever the txdata register is written. The trdy bit is set to 1 after data is transferred from the txdata register into the transmitter shift register, which readies the txdata holding register to receive new data.

A master peripheral should not write to the txdata register until the transmitter is ready for new data. If trdy is 0 and a master peripheral writes new data to the txdata register, a transmit-overrun error occurs and the status register's toe bit is set to 1. In this case, the new data is ignored, and the content of txdata remains unchanged.

As an example, assume that the SPI core is idle (that is, the txdata register and transmit shift register are empty), when a CPU writes a data value into the txdata holding register. The trdy bit is set to 0 momentarily, but after the data in txdata is transferred into the transmitter shift register, trdy returns to 1. The CPU writes a second data value into the txdata register, and again the trdy bit is set to 0. This time the shift register is still busy transferring the original data value, so the trdy bit remains at 0 until the shift operation completes. When the operation completes, the second data value is transferred into the transmitter shift register and the trdy bit is again set to 1.

status Register

The status register consists of bits that indicate status conditions in the SPI core. Each bit is associated with a corresponding interrupt-enable bit in the control register, as discussed in "control Register" on page 10–14.

A master peripheral can read status at any time without changing the value of any bits. Writing status does clear the roe, toe and e bits. Table 10–4 describes the individual bits of the status register.

Table 10–4. status Register Bits

#	Name	Description
3	ROE	**Receive-overrun error** The ROE bit is set to 1 if new data is received while the rxdata register is full (that is, while the RRDY bit is 1). In this case, the new data overwrites the old. Writing to the status register clears the ROE bit to 0.
4	TOE	**Transmitter-overrun error** The TOE bit is set to 1 if new data is written to the txdata register while it is still full (that is, while the TRDY bit is 0). In this case, the new data is ignored. Writing to the status register clears the TOE bit to 0.
5	TMT	**Transmitter shift-register empty** The TMT bit is set to 0 when a transaction is in progress and set to 1 when the shift register is empty.
6	TRDY	**Transmitter ready** The TRDY bit is set to 1 when the txdata register is empty.
7	RRDY	**Receiver ready** The RRDY bit is set to 1 when the rxdata register is full.
8	E	**Error** The E bit is the logical OR of the TOE and ROE bits. This is a convenience for the programmer to detect error conditions. Writing to the status register clears the E bit to 0.

control Register

The control register consists of data bits to control the SPI core's operation. A master peripheral can read control at any time without changing the value of any bits.

Most bits (IROE, ITOE, ITRDY, IRRDY, and IE) in the control register control interrupts for status conditions represented in the status register. For example, bit 1 of status is ROE (receiver-overrun error), and bit 1 of control is IROE, which enables interrupts for the ROE condition. The SPI core asserts an interrupt request when the corresponding bits in status and control are both 1.

The control register bits are shown in Table 10–5.

Table 10–5. control Register Bits

#	Name	Description
3	IROE	Setting IROE to 1 enables interrupts for receive-overrun errors.
4	ITOE	Setting ITOE to 1 enables interrupts for transmitter-overrun errors.
6	ITRDY	Setting ITRDY to 1 enables interrupts for the transmitter ready condition.
7	IRRDY	Setting IRRDY to 1 enables interrupts for the receiver ready condition.
8	IE	Setting IE to 1 enables interrupts for any error condition.
10	SSO	Setting SSO to 1 forces the SPI core to drive its ss_n outputs, regardless of whether a serial shift operation is in progress or not. The slaveselect register controls which ss_n outputs are asserted. sso can be used to transmit or receive data of arbitrary size (in other words, greater than 16 bits).

After reset, all bits of the control register are set to 0. All interrupts are disabled and no ss_n signals are asserted after reset.

slaveselect Register

The slaveselect register is a bit mask for the ss_n signals driven by an SPI master. During a serial shift operation, the SPI master selects only the slave device(s) specified in the slaveselect register.

The slaveselect register is only present when the SPI core is configured in master mode. There is one bit in slaveselect for each ss_n output, as specified by the designer at system generation time. For example, to enable communication with slave device 3, set bit 3 of slaveselect to 1.

A master peripheral can set multiple bits of slaveselect simultaneously, causing the SPI master to simultaneously select multiple slave devices as it performs a transaction. For example, to enable communication with slave devices 1, 5, and 6, set bits 1, 5, and 6 of slaveselect. However, consideration is necessary to avoid signal contention between multiple slaves on their miso outputs.

Upon reset, bit 0 is set to 1, and all other bits are cleared to 0. Thus, after a device reset, slave device 0 is automatically selected.

Referenced Document

This chapter references AN 350: Upgrading Nios Processor Systems to the Nios II Processor.

Document Revision History

Table 10–6 shows the revision history for this chapter.

Table 10–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Chapter 10 was formerly chapter 9. • Added "Referenced Document" on page 10–15.	—
May 2007 v7.1.0	• Chapter 9 was formerly chapter 7. • Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies • Changed old "Avalon switch fabric" term to "system interconnect fabric" • Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface"	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
December 2005 v5.1.1	Changed Avalon ""streaming" terminology to "flow control" based on a change to the *Avalon Interface Specification*.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

Section III. Display Peripherals

This section describes display interface peripherals provided by Altera®. These components provide interfaces to visual display devices for SOPC Builder systems.

See *About This Handbook* for further details.

This section includes the following chapters:

- Chapter 11, Optrex 16207 LCD Controller Core
- Chapter 12, Video Sync Generator and Pixel Converter Cores



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

11. Optrex 16207 LCD Controller Core

NII51019-7.2.0

Core Overview

The Optrex 16207 LCD controller core with Avalon® Interface ("the LCD controller") provides the hardware interface and software driver required for a Nios® II processor to display characters on an Optrex 16207 (or equivalent) 16×2-character LCD panel. Device drivers are provided in the HAL system library for the Nios II processor. Nios II programs access the LCD controller as a character mode device using ANSI C standard library routines, such as `printf()`. The LCD controller is SOPC Builder-ready, and integrates easily into any SOPC Builder-generated system.

The Nios II Embedded Design Suite (EDS) includes an Optrex LCD module and provide several ready-made example designs that display text on the Optrex 16207 via the LCD controller. For details about the Optrex 16207 LCD module, see the manufacturer's *Dot Matrix Character LCD Module User's Manual* available at **www.optrex.com**.

This chapter contains the following sections:

- "Functional Description"
- "Device and Tools Support" on page 11–2
- "Instantiating the Core in SOPC Builder" on page 11–2
- "Software Programming Model" on page 11–2

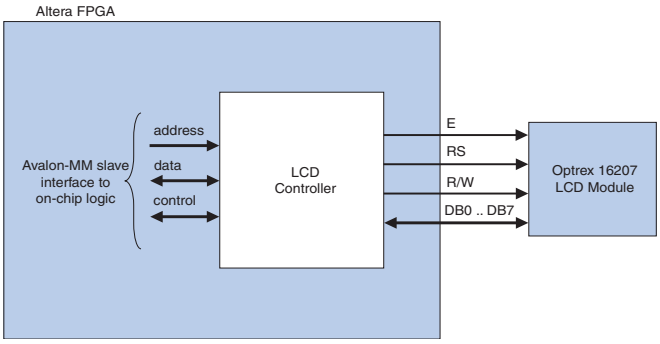
Functional Description

The LCD controller hardware consists of two user-visible components:

- Eleven signals that connect to pins on the Optrex 16207 LCD panel — These signals are defined in the Optrex 16207 data sheet.
  - `E` – Enable (output)
  - `RS` – Register Select (output)
  - `R/W` – Read or Write (output)
  - `DB0` through `DB7` – Data Bus (bidirectional)
- An Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to 4 registers — The HAL device drivers make it unnecessary for users to access the registers directly. Therefore, Altera does not provide details about the register usage. For further details, refer to "Software Programming Model" on page 11–2.

Figure 11–1 shows a block diagram of the LCD controller core.

Figure 11–1. LCD Controller Block Diagram



Device and Tools Support

The LCD controller hardware supports all Altera FPGA families. The LCD controller drivers support the Nios II processor. The drivers do not support the first-generation Nios processor.

Instantiating the Core in SOPC Builder

In SOPC Builder, the LCD controller component has the name Character LCD (16×2, Optrex 16207). The LCD controller does not have any user-configurable settings. The only choice to make in SOPC Builder is whether or not to add an LCD controller to the system. For each LCD controller included in the system, the top-level system module includes the 11 signals that connect to the LCD module.

Software Programming Model

This section describes the software programming model for the LCD controller.

HAL System Library Support

Altera provides HAL system library drivers for the Nios II processor that enable you to access the LCD controller using the ANSI C standard library functions. The Altera-provided drivers integrate into the HAL system library for Nios II systems. The LCD driver is a standard character-mode device, as described in the *Nios II Software Developer's Handbook*. Therefore, using `printf()` is the easiest way to write characters to the display.

The LCD driver requires that the HAL system library include the system clock driver.

Displaying Characters on the LCD

The driver implements VT100 terminal-like behavior on a miniature scale for the 16×2 screen. Characters written to the LCD controller are stored to an 80-column × 2-row buffer maintained by the driver. As characters are written, the cursor position is updated. Visible characters move the cursor position to the right. Any visible characters written to the right of the buffer are discarded. The line feed character (`\n`) moves the cursor down one line and to the left-most column.

The buffer is scrolled up as soon as a printable character is written onto the line below the bottom of the buffer. Rows do not scroll as soon as the cursor moves down to allow the maximum useful information in the buffer to be displayed.

If the visible characters in the buffer will fit on the display, then all characters are displayed. If the buffer is wider than the display, then the display scrolls horizontally to display all the characters. Different lines scroll at different speeds, depending on the number of characters in each line of the buffer.

The LCD driver understands a small subset of ANSI and VT100 escape sequences that can be used to control the cursor position, and clear the display as shown in Table 11–1.

Table 11–1. Escape Sequence Supported by the LCD Controller

Sequence	Meaning
BS (\b)	Moves the cursor to the left by one character.
CR (\r)	Moves the cursor to the start of the current line.
LF (\n)	Moves the cursor to the start of the line and move it down one line.
ESC((\x1B)	Starts a VT100 control sequence.
ESC [<y> ; <x> H	Moves the cursor to the y, x position specified – positions are counted from the top left which is 1;1.
ESC [K	Clears from current cursor position to end of line.
ESC [2 J	Clears the whole screen.

The LCD controller is an output-only device. Therefore, attempts to read from it will return immediately indicating that no characters have been received.

The LCD controller drivers are not included in the system library when the **Reduced device drivers** option is enabled for the system library. If you want to use the LCD controller while using small drivers for other devices, then add the preprocessor option `-DALT_USE_LCD_16207` to the preprocessor options.

Software Files

The LCD controller is accompanied by the following software files. These files define the low-level interface to the hardware and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_lcd_16207_regs.h** — This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_lcd_16207.h, altera_avalon_lcd_16207.c** — These files implement the LCD controller device drivers for the HAL system library.

Register Map

The HAL device drivers make it unnecessary for you to access the registers directly. Therefore, Altera does not publish details about the register map. For more information, the **altera_avalon_lcd_16207_regs.h** file describes the register map, and the *Dot Matrix Character LCD Module User's Manual* from Optrex describes the register usage.

Interrupt Behavior

The LCD controller does not generate interrupts. However, the LCD driver's text scrolling feature relies on the HAL system clock driver, which uses interrupts for timing purposes.

Referenced Document

This chapter references the Nios II Software Developer's Handbook.

Document Revision History

Table 11–2 shows the revision history for this chapter.

Table 11–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 11 was formerly chapter 10.	—
May 2007 v7.1.0	• Chapter 10 was formerly chapter 8. • Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies • Changed old “Avalon switch fabric” term to “system interconnect fabric” • Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	Chapter title changed, but no change in content from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.0	Initial release.	—

12. Video Sync Generator and Pixel Converter Cores

QII55006-7.2.0

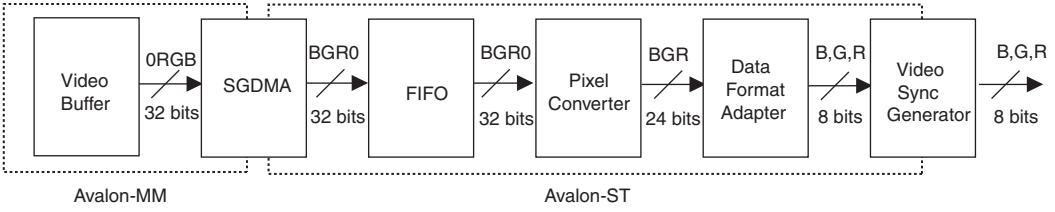
Core Overview

The video sync generator core accepts a continuous stream of pixel data in RGB format, and outputs the data to an off-chip display controller with proper timing. You can configure the video sync generator core to support different display resolutions and synchronization timings.

The pixel converter core transforms the pixel data to the format required by the video sync generator. Figure 12–1 shows a typical placement of the video sync generator and pixel converter cores in a system.

In this example, the video buffer stores the pixel data in 32-bit unpacked format. The extra byte in the pixel data is discarded by the pixel converter core before the data is serialized and sent to the video sync generator core.

Figure 12–1. Typical Placement in a System



The video sync generator and pixel converter cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system.

These cores are deployed in the Nios II Embedded Software Evaluation Kit (EEK), which includes an LCD display daughtercard assembly attached via an HSMC connector.

This chapter contains the following sections:

- "Video Sync Generator" on page 12–2
- "Pixel Converter" on page 12–5
- "Device and Tools Support" on page 12–6
- "Hardware Simulation Considerations" on page 12–7

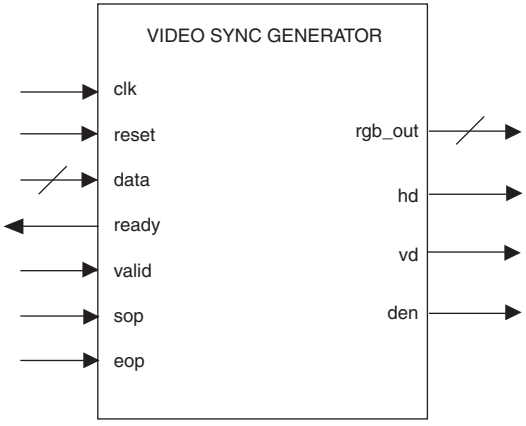
Video Sync Generator

This section describes the hardware structure and functionality of the video sync generator core.

Functional Description

The video sync generator core adds horizontal and vertical synchronization signals to the pixel data that comes through its Avalon-ST input interface and outputs the data to an off-chip display controller. No processing or validation is performed on the pixel data. Figure 12–2 shows a block diagram of the video sync generator.

Figure 12–2. Video Sync Generator Block Diagram



You can configure various aspects of the core and its Avalon-ST interface to suit your requirements. You can specify the data width, number of beats required to transfer each pixel and synchronization signals. See "Instantiating the Core in SOPC Builder" on page 12–3 for more information on the available options.

To ensure incoming pixel data is sent to the display controller with correct timing, the video sync generator core must synchronize itself to the first pixel in a frame. The first active pixel is indicated by an `sop` pulse.

The video sync generator core expects continuous streams of pixel data at its input interface and assumes that each incoming packet contains the correct number of pixels (Number of rows * Number of columns). Data starvation disrupts synchronization and results in unexpected output on the display.

Instantiating the Core in SOPC Builder

Use the MegaWizard® interface for the video sync generator core in SOPC Builder to configure the core. Table 12–1 lists the available parameters in the MegaWizard interface.

Table 12–1. Video Sync Generator Parameters

Parameter Name	Description
Data Stream Bit Width	The width of the inbound and outbound data.
Beats Per Pixel	The number of beats required to transfer one pixel. Valid values are 1 and 3. This parameter, when multiplied by **Data Stream Bit Width** must be equal to the total number of bits in one pixel.
Number of Columns	The number of active pixels in each line.
Number of Rows	The number of active scan lines in each video frame.
Horizontal Blank Pixels	The number of blanking pixels that preceed the active pixels. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Horizontal Front Porch Pixels	The number of blanking pixels that follow the active pixels. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Vertical Blank Lines	The number of blanking lines that preceed the active lines. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Vertical Front Porch Pixels	The number of blanking lines that follow the active lines. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Total Horizontal Scan Pixels	The total number of pixels in one line. The value is the sum of the following parameters: **Number of Columns**, **Horizontal Blank Pixel**, and **Horizontal Front Porch Pixels**.
Total Vertical Scan Lines	The total number of lines in one video frame. The value is the sum of the following parameters: **Number of Rows**, **Vertical Blank Lines**, and **Vertical Front Porch Lines**.

Signals

Table 12–2 lists the input and output signals for the video sync generator core.

Table 12–2. Video Sync Generator Core Signals (Part 1 of 2)

Signal Name	Width (Bits)	Direction	Description
Global Signals			
`clk`	1	Input	System clock.
`reset`	1	Input	System reset.
Avalon-ST Signals			

Table 12–2. Video Sync Generator Core Signals (Part 2 of 2)

Signal Name	Width (Bits)	Direction	Description
data	Variable-width	Input	Incoming pixel data. The datawidth is determined by the parameter **Data Stream Bit Width**.
ready	1	Output	This signal is asserted when the video sync generator is ready to receive the pixel data.
valid	1	Input	This signal is not used by the video sync generator core because the core always expects valid pixel data on the next clock cycle after the ready signal is asserted.
sop	1	Input	Start-of-packet. This signal is asserted when the first pixel is received.
eop	1	Input	End-of-packet. This signal is asserted when the last pixel is received.
LCD Output Signals			
rgb_out	Variable-width	Output	Display data. The datawidth is determined by the parameter **Data Stream Bit Width**.
hd	1	Output	Horizontal synchronization pulse for display.
vd	1	Output	Vertical synchronization pulse for display.
den	1	Output	This signal is asserted when the video sync generator core outputs valid data for display.

Timing Diagrams

The horizontal and vertical synchronization timings are determined by the parameters setting. Figure 12–3 shows the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **Beats Per Pixel** are set to 8 and 3, respectively.

Figure 12–3. Horizontal Synchronization Timing—8 bits DataWidth and 3 Beats Per Pixel

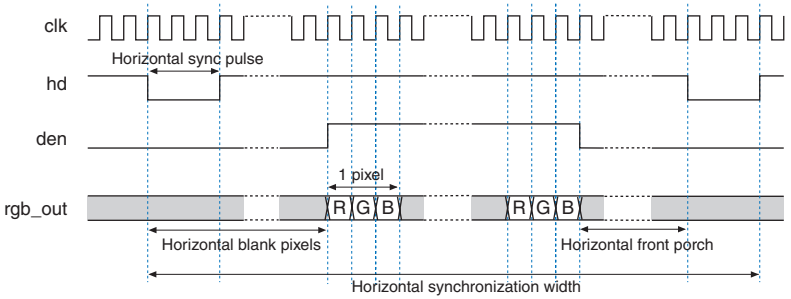


Figure 12–4 shows the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **Beats Per Pixel** are set to 24 and 1, respectively.

Figure 12–4. Horizontal Synchronization Timing—24 bits Datawidth and 1 Beat Per Pixel

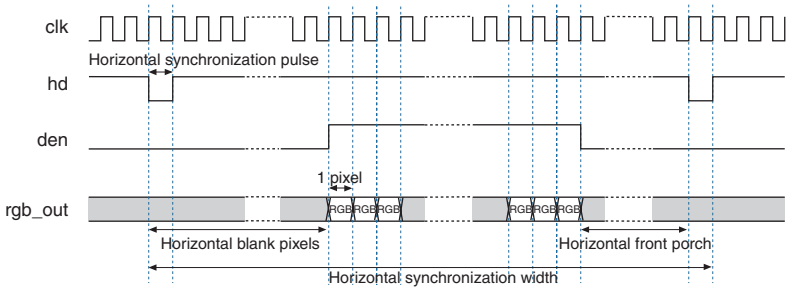
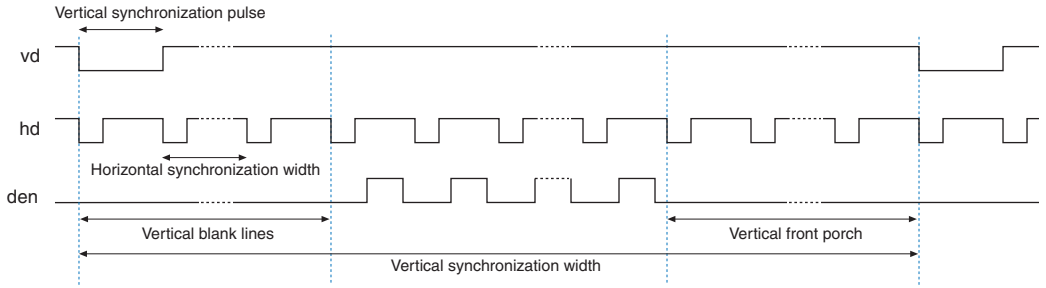


Figure 12–5 shows the vertical synchronization timing.

Figure 12–5. Vertical Synchronization Timing



Pixel Converter

This section describes the hardware structure and functionality of the pixel converter core.

Functional Description

The pixel converter core receives pixel data on its Avalon-ST input interface and transforms the pixel data to the format required by the video sync generator. The least significant byte of the 32-bit wide pixel data is removed and the remaining 24 bits are wired directly to the core's Avalon-ST output interface.

Instantiating the Core in SOPC Builder

Use the MegaWizard interface for the pixel converter core in SOPC Builder to add the core to a system. There are no user-configurable settings for this core.

Signals

Table 12–3 lists the input and output signals for the pixel converter core.

Table 12–3. Pixel Converter Input Interface Signals

Signal Name	Width (Bits)	Direction	Description
Global Signals			
clk	1	Input	Not in use.
reset_n	1	Input	
Avalon-ST Signals			
data_in	32	Input	Incoming pixel data. Contains four 8-bit symbols that are tranferred in 1 beat.
data_out	24	Output	Output data. Contains three 8-bit symbols that are transferred in 1 beat.
sop_in	1	Input	Wired directly to the corresponding output signals.
eop_in	1	Input	
ready_in	1	Input	
valid_in	1	Input	
empty_in	1	Input	
sop_out	1	Output	Wired directly from the input signals.
eop_out	1	Output	
ready_out	1	Output	
valid_out	1	Output	
empty_out	1	Output	

Device and Tools Support

The video sync generator and pixel converter cores support all Altera device families.

Hardware Simulation Considerations

For a typical 60 Hz refresh rate, set the simulation length for the video sync generator core to at least 16.7 ms to get a full video frame. Depending on the size of the video frame, simulation may take a very long time to complete.

Referenced Document

This chapter references the *Avalon Streaming Interface Specification*.

Document Revision History

Table 12–4 shows the revision history for this chapter.

Table 12–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Initial release.	—

Section IV. Multiprocessor Coordination Peripherals

This section describes multiprocessor coordination peripherals provided by Altera® for SOPC Builder systems. These components provide reliable mechanisms for multiple Nios® II processors to communicate with each other, and coordinate operations.

See *About This Handbook* for further details.

This section includes the following chapters:

- Chapter 13, Mutex Core
- Chapter 14, Mailbox Core



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

13. Mutex Core

NII51020-7.2.0

Core Overview

Multiprocessor environments can use the mutex core with Avalon® interface to coordinate accesses to a shared resource. The mutex core provides a protocol to ensure mutually exclusive ownership of a shared resource.

The mutex core provides a hardware-based atomic test-and-set operation, allowing software in a multiprocessor environment to determine which processor owns the mutex. The mutex core can be used in conjunction with shared memory to implement additional interprocessor coordination features, such as mailboxes and software mutexes.

The mutex core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera provides device drivers for the Nios II processor to enable use of the hardware mutex.

The mutex core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- Functional Description
- "Device and Tools Support" on page 13–2
- "Instantiating the Core in SOPC Builder" on page 13–2
- "Software Programming Model" on page 13–2
- "Mutex API" on page 13–4

Functional Description

The mutex core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to two memory-mapped, 32-bit registers. Table 13–1 shows the registers.

Table 13–1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description		
			31 … 16	15 … 1	0
0	mutex	RW	OWNER	VALUE	
1	reset	RW	–	–	RESET

The mutex core has the following basic behavior. This description assumes there are multiple processors accessing a single mutex core, and each processor has a unique identifier (ID).

- When the VALUE field is `0x0000`, the mutex is available (i.e, unlocked). Otherwise, the mutex is unavailable (i.e., locked).
- The `mutex` register is always readable. A processor (or any Avalon-MM master peripheral) can read the `mutex` register to determine its current state.
- The `mutex` register is writable only under specific conditions. A write operation changes the `mutex` register only if one or both of the following conditions is true:
  - The VALUE field of the `mutex` register is zero.
  - The OWNER field of the `mutex` register matches the OWNER field in the data to be written.
- A processor attempts to acquire the mutex by writing its ID to the OWNER field, and writing a non-zero value to VALUE. The processor then checks if the acquisition succeeded by verifying the OWNER field.
- After system reset, the RESET bit in the `reset` register is high. Writing a one to this bit clears it.

Device and Tools Support

The mutex core supports all Altera device families supported by SOPC Builder, and provides device drivers for the Nios II hardware abstraction layer (HAL) system library.

Instantiating the Core in SOPC Builder

Hardware designers use the MegaWizard® interface for the mutex core in SOPC Builder to specify the core's hardware features. The MegaWizard interface provides the following options:

- **Initial Value**—the initial contents of the VALUE field after reset. If the **Initial Value** setting is non-zero, you must also specify **Initial Owner**.
- **Initial Owner**—the initial contents of the OWNER field after reset. When **Initial Owner** is specified, this owner must release the mutex before it can be acquired by another owner.

Software Programming Model

The following sections describe the software programming model for the mutex core, such as the software constructs used to access the hardware. For Nios II processor users, Altera provides routines to access the mutex core hardware. These functions are specific to the mutex core and directly manipulate low-level hardware. The mutex core cannot be accessed via the HAL API or the ANSI C standard library. In Nios II processor systems, a processor locks the mutex by writing the value of its `cpuid` control register to the OWNER field of the `mutex` register.

Software Files

Altera provides the following software files accompanying the mutex core:

- **altera_avalon_mutex_regs.h**—this file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_mutex.h—**this file defines data structures and functions to access the mutex core hardware.
- **altera_avalon_mutex.c**—this file contains the implementations of the functions to access the mutex core

Hardware Mutex

This section describes the low-level software constructs for manipulating the mutex core hardware.

The file **altera_avalon_mutex.h** declares a structure `alt_mutex_dev` that represents an instance of a mutex device. It also declares functions for accessing the mutex hardware structure, listed in Table 13–2.

Table 13–2. Hardware Mutex Functions

Function Name	Description
`altera_avalon_mutex_open()`	Claims a handle to a mutex, enabling all the other functions to access the mutex core.
`altera_avalon_mutex_trylock()`	Tries to lock the mutex. Returns immediately if it fails to lock the mutex.
`altera_avalon_mutex_lock()`	Locks the mutex. Will not return until it has successfully claimed the mutex.
`altera_avalon_mutex_unlock()`	Unlocks the mutex.
`altera_avalon_mutex_is_mine()`	Determines if this CPU owns the mutex.
`altera_avalon_mutex_first_lock()`	Tests whether the mutex has been released since reset.

These routines coordinate access to the software mutex structure using a hardware mutex core. For a complete description of each function, see section "Mutex API" on page 13–4.

Example 13–1 demonstrates opening a mutex device handle and locking a mutex:

Example 13–1. Opening and locking a mutex

```
#include <altera_avalon_mutex.h>

/* get the mutex device handle */
alt_mutex_dev* mutex = altera_avalon_mutex_open( "/dev/mutex" );

/* acquire the mutex, setting the value to one */
altera_avalon_mutex_lock( mutex, 1 );

/*
 * Access a shared resource here.
 */

/* release the lock */
altera_avalon_mutex_unlock( mutex );
```

Mutex API

This section describes the application programming interface (API) for the mutex core.

altera_avalon_mutex_is_mine()

Prototype:	`int altera_avalon_mutex_is_mine(alt_mutex_dev* dev)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to test.
Returns:	Returns non zero if the mutex is owned by this CPU.
Description:	`altera_avalon_mutex_is_mine()` determines if this CPU owns the mutex.

altera_avalon_mutex_first_lock()

Prototype:	`int altera_avalon_mutex_first_lock(alt_mutex_dev* dev)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to test.
Returns:	Returns 1 if this mutex has not been released since reset, otherwise returns 0.
Description:	`altera_avalon_mutex_first_lock()` determines whether this mutex has been released since reset.

altera_avalon_mutex_lock()

Prototype:	`void altera_avalon_mutex_lock(alt_mutex_dev* dev, alt_u32 value)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to acquire. `value`—the new value to write to the mutex.
Returns:	–
Description:	`altera_avalon_mutex_lock()` is a blocking routine that acquires a hardware mutex, and at the same time, loads the mutex with the `value` parameter.

altera_avalon_mutex_open()

Prototype:	`alt_mutex_dev* alt_hardware_mutex_open(const char* name)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`name`—the name of the mutex device to open.
Returns:	A pointer to the mutex device structure associated with the supplied name, or NULL if no corresponding mutex device structure was found.
Description:	`altera_avalon_mutex_open()` retrieves a pointer to a hardware mutex device structure.

altera_avalon_mutex_trylock()

Prototype:	`int altera_avalon_mutex_trylock(alt_mutex_dev* dev, alt_u32 value)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to lock. `value`—the new value to write to the mutex.
Returns:	Zero if the mutex was successfully locked, or non zero if the mutex was not locked.
Description:	`altera_avalon_mutex_trylock()` tries once to lock the hardware mutex, and returns immediately.

altera_avalon_mutex_unlock()

Prototype:	`void altera_avalon_mutex_unlock(alt_mutex_dev* dev)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to unlock.
Returns:	-
Description:	`altera_avalon_mutex_unlock()` releases a hardware mutex device. Upon release, the value stored in the mutex is set to zero. If the caller does not hold the mutex, the behavior of this function is undefined.

Document Revision History

Table 13–3 shows the revision history for this chapter.

Table 13–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 13 was formerly chapter 11.	—
May 2007 v7.1.0	• Chapter 11 was formerly chapter 9. • Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies • Changed old “Avalon switch fabric” term to “system interconnect fabric” • Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
December 2004 v1.0	Initial release.	—

14. Mailbox Core

NII53001-7.2.0

Core Overview

Multiprocessor environments can use the mailbox core with Avalon® interface to send messages between processors.

The mailbox core contains mutexes to ensure that only one processor modifies the mailbox contents at a time. The mailbox core must be used in conjunction with a separate shared memory that is used for storing the actual messages.

The mailbox core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera® provides device drivers for the Nios II processor. The mailbox core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device and Tools Support" on page 14–2
- "Instantiating the Core in SOPC Builder" on page 14–2
- "Software Programming Model" on page 14–3
- "Mailbox API" on page 14–6

Functional Description

The mailbox core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to four memory-mapped, 32-bit registers. Table 14–1 shows the registers.

Table 14–1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description		
			31 … 16	15 … 1	0
0	mutex0	RW	OWNER	VALUE	
1	reset0	RW	–	–	RESET
2	mutex1	RW	OWNER	VALUE	
3	reset1	RW	–	–	RESET

The mailbox component contains two mutexes: One to ensure unique write access to shared memory and one to ensure unique read access from shared memory. The mailbox core is used in conjunction with a separate memory in the system that is shared among multiple processors.

Mailbox functionality using the mutexes and memory is implemented entirely in the software. Refer to "Software Programming Model" on page 14–3 for details about how to use the mailbox core in software.



For a detailed description of the mutex hardware operation, refer to the Mutex Core chapter in volume 5 of the *Quartus II Handbook*.

Device and Tools Support

The mailbox core supports all Altera device families supported by SOPC Builder, and provides device drivers for the Nios II hardware abstraction layer (HAL) system library.

Instantiating the Core in SOPC Builder

Hardware designers instantiate and configure the mailbox core in an SOPC Builder system using the following process:

1. Decide which processors will share the mailbox.
2. On the SOPC Builder **System Contents** tab, instantiate a memory component to serve as the mailbox buffer. Any RAM can be used as the mailbox buffer. The mailbox buffer can share space in an existing memory, such as program memory; it does not require a dedicated memory.
3. On the SOPC Builder **System Contents** tab, instantiate the mailbox component. The mailbox MegaWizard® interface presents the following options:
   - **Memory module**—Specifies which memory to use for the mailbox buffer. If the **Memory module** list does not contain the desired shared memory, the memory is not connected in the system correctly. Refer to Step 4 on page 14–3.
   - **CPUs available with this memory**—Shows all the processors that can share the mailbox. This field is always read-only. Use it to verify that the processor connections are correct. If a processor that needs to share the mailbox is missing from the list, refer to Step 4 on page 14–3.
   - **Shared mailbox memory offset**—Specifies an offset into the memory. The mailbox message buffer starts at this offset.
   - **Mailbox size (bytes)**—Specifies the number of bytes to use for the mailbox message buffer. The Nios II driver software provided by Altera uses eight bytes of overhead to implement the mailbox functionality. For a mailbox capable of passing only

one message at a time, **Mailbox size (bytes)** must be at least 12 bytes.

- **Maximum available bytes**—Specifies the number of bytes in the selected memory available for use as the mailbox message buffer. This field is always read-only.

4. If not already connected, make component connections on the SOPC Builder **System Contents** tab.

   a. Connect each processor's data bus master port to the mailbox slave port.

   b. Connect each processor's data bus master port to the shared mailbox memory.

Software Programming Model

The following sections describe the software programming model for the mailbox core, such as the software constructs used to access the hardware. For Nios II processor users, Altera provides routines to access the mailbox core hardware. These functions are specific to the mailbox core and directly manipulate low-level hardware.

The mailbox software programming model has the following characteristics and assumes there are multiple processors accessing a single mailbox core and a shared memory:

- Each mailbox message is one 32-bit word.
- There is a predefined address range in shared memory dedicated to storing messages. The size of this address range imposes a maximum limit on the number of messages pending.
- The mailbox software implements a message FIFO between processors. Only one processor can write to the mailbox at a time, and only one processor can read from the mailbox at a time, ensuring message integrity.
- The software on both the sending and receiving processors must agree on a protocol for interpreting mailbox messages. Typically, processors treat the message as a pointer to a structure in shared memory.
- The sending processor can post messages in succession, up to the limit imposed by the size of the message address range.
- When messages exist in the mailbox, the receiving processor can read messages. The receiving processor can block until a message appears, or it can poll the mailbox for new messages.
- Reading the message removes the message from the mailbox.

Software Files

Altera provides the following software files accompanying the mailbox core hardware:

- **altera_avalon_mailbox_regs.h**—Defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_mailbox.h**—Defines data structures and functions to access the mailbox core hardware.
- **altera_avalon_mailbox.c**—Contains the implementations of the functions to access the mailbox core.

Programming with the Mailbox Core

This section describes the software constructs for manipulating the mailbox core hardware.

The file **altera_avalon_mailbox.h** declares a structure `alt_mailbox_dev` that represents an instance of a mailbox device. It also declares functions for accessing the mailbox hardware structure, listed in Table 14–2. For a complete description of each function, refer to "Mailbox API" on page 14–6.

Table 14–2. Mailbox API Functions

Function Name	Description
`altera_avalon_mailbox_close()`	Closes the handle to a mailbox.
`altera_avalon_mailbox_get()`	Returns a message if one is present, but does not block waiting for a message.
`altera_avalon_mailbox_open()`	Claims a handle to a mailbox, enabling all the other functions to access the mailbox core.
`altera_avalon_mailbox_pend()`	Blocks waiting for a message to be in the mailbox.
`altera_avalon_mailbox_post()`	Posts a message to the mailbox.

Example 14–1 demonstrates writing to and reading from a mailbox. For this example, assume that the hardware system has two processors communicating via mailboxes. The system includes two mailbox cores, which provide two-way communication between the processors.

Example 14–1. Example: Writing to and Reading from a Mailbox

```
#include <stdio.h>
#include "altera_avalon_mailbox.h"

int main()
{
  alt_u32 message = 0;
  alt_mailbox_dev* send_dev, recv_dev;
  /* Open the two mailboxes between this processor and another */
  send_dev = altera_avalon_mailbox_open("/dev/mailbox_0");
  recv_dev = altera_avalon_mailbox_open("/dev/mailbox_1");

  while(1)
  {
    /* Send a message to the other processor */
    altera_avalon_mailbox_post(send_dev, message);

    /* Wait for the other processor to send a message back */
    message = altera_avalon_mailbox_pend(recv_dev);

  }

  return 0;
}
```

Mailbox API

This section describes the application programming interface (API) for the mailbox core.

altera_avalon_mailbox_close()

Prototype:	`void altera_avalon_mailbox_close (alt_mailbox_dev* dev);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`dev`—the mailbox to close.
Returns:	–
Description:	`altera_avalon_mailbox_close()` closes the mailbox.

altera_avalon_mailbox_get()

Prototype:	`alt_u32 altera_avalon_mailbox_get (alt_mailbox_dev* dev, int* err);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`dev`—the mailbox handle, `err`—pointer to an error code that is returned.
Returns:	Returns a message if one is available in the mailbox, otherwise returns 0. The value pointed to by `err` is 0 if the message was read correctly, or `EWOULDBLOCK` if there is no message to read.
Description:	`altera_avalon_mailbox_get()` returns a message if one is present, but does not block waiting for a message.

altera_avalon_mailbox_open()

Prototype:	`alt_mailbox_dev* altera_avalon_mailbox_open (const char* name);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`name`—the name of the mailbox device to open.
Returns:	Returns a handle to the mailbox, or NULL if this mailbox does not exist.
Description:	`altera_avalon_mailbox_open()` opens a mailbox.

altera_avalon_mailbox_pend()

Prototype:	`alt_u32 altera_avalon_mailbox_pend (alt_mailbox_dev* dev);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`dev`—the mailbox device to read a message from.
Returns:	Returns the message.
Description:	`altera_avalon_mailbox_pend()` is a blocking routine that waits for a message to appear in the mailbox and then reads it.

altera_avalon_mailbox_post()

Prototype:	`int altera_avalon_mailbox_post (alt_mailbox_dev* dev, alt_u32 msg);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`dev`—the mailbox device to post a message to `msg`—the value to post.
Returns:	Returns 0 on success, or `EWOULDBLOCK` if the mailbox is full.
Description:	`altera_avalon_mailbox_post()` posts a message to the mailbox.

Referenced Document

This chapter references the *Mutex Core* chapter in volume 5 of the *Quartus II Handbook*

Document Revision History

Table 14–3 shows the revision history for this chapter.

Table 14–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 14 was formerly chapter 12.	—
May 2007 v7.1.0	• Chapter 12 was formerly chapter 10. • Revised "Instantiating the Core in SOPC Builder" on page 14–2 to reflect the GUI changing from the More Settings tab to the MegaWizard interface. • Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies. • Changed old "Avalon switch fabric" term to "system interconnect fabric." • Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface."	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	Initial release.	—

Section V. Other Memory-Mapped Peripherals

This section describes other peripherals provided by Altera® for SOPC Builder systems.

See *About This Handbook* for further details.

This section includes the following chapters:

- Chapter 15, PIO Core
- Chapter 16, Timer Core
- Chapter 17, System ID Core
- Chapter 18, PLL Core
- Chapter 19, Performance Counter Core



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

15. PIO Core

NII51007-7.2.0

Core Overview

The parallel input/output (PIO) core with Avalon® interface provides a memory-mapped interface between an Avalon Memory-Mapped (Avalon-MM) slave port and general-purpose I/O ports. The I/O ports connect either to on-chip user logic, or to I/O pins that connect to devices external to the FPGA.

The PIO core provides easy I/O access to user logic or external devices in situations where a "bit banging" approach is sufficient. Some example uses are:

- Controlling LEDs
- Acquiring data from switches
- Controlling display devices
- Configuring and communicating with off-chip devices, such as application-specific standard products (ASSP)

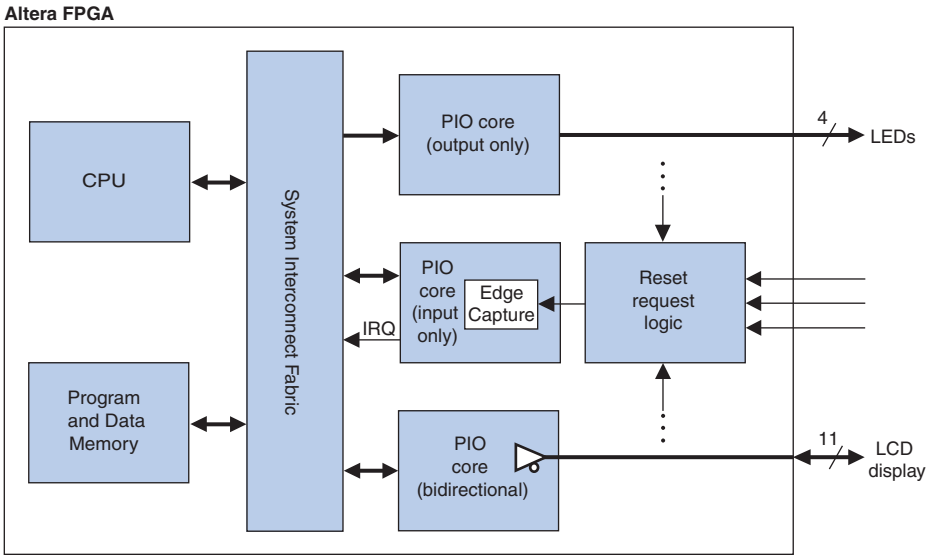
The PIO core interrupt request (IRQ) output can assert an interrupt based on input signals. The PIO core is SOPC Builder ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- Functional Description
- "Example Configurations" on page 15–4
- "Instantiating the PIO Core in SOPC Builder" on page 15–5
- "Device and Tools Support" on page 15–6
- "Software Programming Model" on page 15–6

Functional Description

Each PIO core can provide up to 32 I/O ports. An intelligent host such as a microprocessor controls the PIO ports by reading and writing the register-mapped Avalon-MM interface. Under control of the host, the PIO core captures data on its inputs and drives data to its outputs. When the PIO ports are connected directly to I/O pins, the host can tristate the pins by writing control registers in the PIO core. Figure 15–1 shows an example of a processor-based system that uses multiple PIO cores to blink LEDs, capture edges from on-chip reset-request control logic, and control an off-chip LCD display.

Figure 15–1. An Example System Using Multiple PIO Cores



When integrated into an SOPC Builder-generated system, the PIO core has two user-visible features:

- A memory-mapped register space with four registers: `data`, `direction`, `interruptmask`, and `edgecapture`.
- 1 to 32 I/O ports.

The I/O ports can be connected to logic inside the FPGA, or to device pins that connect to off-chip devices. The registers provide an interface to the I/O ports via the Avalon-MM interface. See Table 15–2 on page 15–7 for a description of the registers. Some registers are not necessary in certain hardware configurations, in which case the unnecessary registers do not exist. Reading a non-existent register returns an undefined value, and writing a non-existent register has no effect.

Data Input and Output

The PIO core I/O ports can connect to either on-chip or off-chip logic. The core can be configured with inputs only, outputs only, or both inputs and outputs. If the core will be used to control bidirectional I/O pins on the device, the core provides a bidirectional mode with tristate control.

The hardware logic is separate for reading and writing the data register. Reading the data register returns the value present on the input ports (if present). Writing data affects the value driven to the output ports (if present). These ports are independent; reading the data register does not return previously-written data.

Edge Capture

The PIO core can be configured to capture edges on its input ports. It can capture low-to-high transitions, high-to-low transitions, or both. Whenever an input detects an edge, the condition is indicated in the `edgecapture` register. The type of edges to detect is specified at system generation time, and cannot be changed via the registers.

IRQ Generation

The PIO core can be configured to generate an IRQ on certain input conditions. The IRQ conditions can be either:

- *Level-sensitive*—The PIO core hardware can detect a high level. A `NOT` gate can be inserted external to the core to provide negative sensitivity.
- *Edge-sensitive*—The core's edge capture configuration determines which type of edge causes an IRQ

Interrupts are individually maskable for each input port. The interrupt mask determines which input port can generate interrupts.

Example Configurations

Figure 15–2 shows a block diagram of the PIO core configured with input and output ports, as well as support for IRQs.

Figure 15–2. PIO Core with Input Ports, Output Ports and IRQ Support

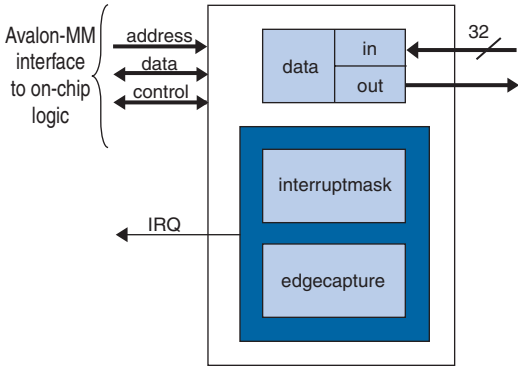
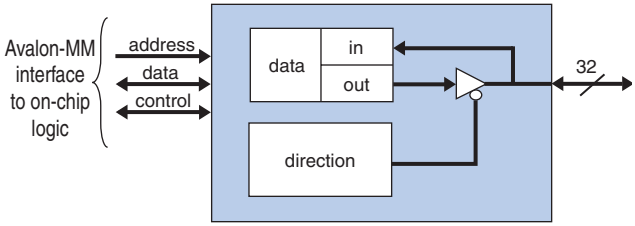


Figure 15–3 shows a block diagram of the PIO core configured in bidirectional mode, without support for IRQs.

Figure 15–3. PIO Core with Bidirectional Ports



Avalon-MM Interface

The PIO core's Avalon-MM interface consists of a single Avalon-MM slave port. The slave port is capable of fundamental Avalon-MM read and write transfers. The Avalon-MM slave port provides an IRQ output so that the core can assert interrupts.

Instantiating the PIO Core in SOPC Builder

Designers use the MegaWizard® interface for the PIO core in SOPC Builder to configure the hardware feature set. The following sections describe the available options.

The MegaWizard interface has two tabs, **Basic Settings** and **Input Options**.

Basic Settings

The **Basic Settings** page allows the designer to specify the width and direction of the I/O ports.

- The **Width** setting can be any integer value between 1 and 32. For a value of *n*, the I/O ports become *n*-bits wide.
- The **Direction** setting has four options, as shown in Table 15–1.

Table 15–1. Direction Settings

Setting	Description
Bidirectional (tristate) ports	In this mode, each PIO bit shares one device pin for driving and capturing data. The direction of each pin is individually selectable. To tristate an FPGA I/O pin, set the direction to input.
Input ports only	In this mode the PIO ports can capture input only.
Output ports only	In this mode the PIO ports can drive output only.
Both input and output ports	In this mode, the input and output ports buses are separate, unidirectional buses of *n* bits wide.

Input Options

The **Input Options** page allows the designer to specify edge-capture and IRQ generation settings. The **Input Options** page is not available when **Output ports only** is selected on the **Basic Settings** page.

Edge Capture Register

Synchronously Capture
When **Synchronously capture** is on, the PIO core contains the edge capture register, `edgecapture`. The user must further specify what type of edge(s) to detect:

- **Rising Edge**
- **Falling Edge**
- **Either Edge**

The edge capture register allows the core to detect and (optionally) generate an interrupt when an edge of the specified type occurs on an input port.

When **Synchronously capture** is off, the `edgecapture` register does not exist.

Enable Bit Clearing for Edge Capture Register
Turning on **Enable bit-clearing for edge capture register** allows you to clear individual bit(s) in the edge capture register. To clear a given bit, write 1 to the bit in the edge capture register. Example—To clear bit 6 in the edge capture register, write 01000000 to the register.

Interrupt

When **Generate IRQ** is on, the PIO core is able to assert an IRQ output when a specified event occurs on input ports. The user must further specify the cause of an IRQ event:

- **Level—**The core generates an IRQ whenever a specific input is high and interrupts are enabled for that input in the `interruptmask` register.
- **Edge**—The core generates an IRQ whenever a specific bit in the edge capture register is high and interrupts are enabled for that bit in the `interruptmask` register.

When **Generate IRQ** is off, the `interruptmask` register does not exist.

Device and Tools Support

The PIO core supports all Altera® FPGA families.

Software Programming Model

This section describes the software programming model for the PIO core, including the register map and software constructs used to access the hardware. For Nios® II processor users, Altera provides the HAL system library header file that defines the PIO core registers. The PIO core does not match the generic device model categories supported by the HAL, so it cannot be accessed via the HAL API or the ANSI C standard library.

The Nios II Embedded Design Suite (EDS) provides several example designs that demonstrate usage of the PIO core. In particular, the **count_binary.c** example uses the PIO core to drive LEDs, and detect button presses using PIO edge-detect interrupts.

Software Files

The PIO core is accompanied by one software file, **altera_avalon_pio_regs.h**. This file defines the core's register map, providing symbolic constants to access the low-level hardware.

Legacy SDK Routines

The PIO core is supported by the legacy SDK routines for the first-generation Nios processor. For details about these routines, refer to the PIO documentation that accompanied the first-generation Nios processor.



For details about upgrading programs based on the legacy SDK to the HAL system library API, refer to *AN 350: Upgrading Nios Processor Systems to the Nios II Processor*.

Register Map

An Avalon-MM master peripheral, such as a CPU, controls and communicates with the PIO core via the four 32-bit registers, shown in Table 15–2. The table assumes that the PIO core's I/O ports are configured to a width of *n* bits.

Table 15–2. Register Map for the PIO Core

Offset	Register Name		R/W	(n-1)	…	2	1	0
0	`data`	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	`direction` *(1)*		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	`interruptmask` *(1)*		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	`edgecapture` *(1)*, *(2)*		R/W	Edge detection for each input port.				

Notes to Table 15–2:

(1) This register may not exist, depending on the hardware configuration. If a register is not present, reading the register returns an undefined value, and writing the register has no effect.

(2) Writing any value to `edgecapture` clears all bits to 0.

data Register

Reading from `data` returns the value present at the input ports. If the PIO core hardware is configured in output-only mode, reading from `data` returns an undefined value.

Writing to data stores the value to a register that drives the output ports. If the PIO core hardware is configured in input-only mode, writing to data has no effect. If the PIO core hardware is in bidirectional mode, the registered value appears on an output port only when the corresponding bit in the direction register is set to 1 (output).

direction Register

The direction register controls the data direction for each PIO port, assuming the port is bidirectional. When bit *n* in direction is set to 1, port *n* drives out the value in the corresponding bit of the data register.

The direction register only exists when the PIO core hardware is configured in bidirectional mode. The mode (input, output, or bidirectional) is specified at system generation time, and cannot be changed at runtime. In input-only or output-only mode, the direction register does not exist. In this case, reading direction returns an undefined value, writing direction has no effect.

After reset, all bits of direction are 0, so that all bidirectional I/O ports are configured as inputs. If those PIO ports are connected to device pins, the pins are held in a high-impedance state. In bi-directional mode, to change the direction of the PIO port re-program the direction register.

interruptmask Register

Setting a bit in the interruptmask register to 1 enables interrupts for the corresponding PIO input port. Interrupt behavior depends on the hardware configuration of the PIO core. See "Interrupt Behavior" on page 15–9.

The interruptmask register only exists when the hardware is configured to generate IRQs. If the core cannot generate IRQs, reading interruptmask returns an undefined value, and writing to interruptmask has no effect.

After reset, all bits of interruptmask are zero, so that interrupts are disabled for all PIO ports.

edgecapture Register

Bit *n* in the edgecapture register is set to 1 whenever an edge is detected on input port *n*. An Avalon-MM master peripheral can read the edgecapture register to determine if an edge has occurred on any of the PIO input ports. Writing any value to edgecapture clears all bits in the register.

The type of edge(s) to detect is fixed in hardware at system generation time. The `edgecapture` register only exists when the hardware is configured to capture edges. If the core is not configured to capture edges, reading from `edgecapture` returns an undefined value, and writing to `edgecapture` has no effect.

Interrupt Behavior

The PIO core outputs a single IRQ signal that can connect to any master peripheral in the system. The master can read either the `data` register or the `edgecapture` register to determine which input port caused the interrupt.

When the hardware is configured for level-sensitive interrupts, the IRQ is asserted whenever corresponding bits in the `data` and `interruptmask` registers are 1. When the hardware is configured for edge-sensitive interrupts, the IRQ is asserted whenever corresponding bits in the `edgecapture` and `interruptmask` registers are 1. The IRQ remains asserted until explicitly acknowledged by disabling the appropriate bit(s) in `interruptmask`, or by writing to `edgecapture`.

Software Files

The PIO core is accompanied by the following software file. This file provide low-level access to the hardware. Application developers should not modify the file.

- **altera_avalon_pio_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used by device driver functions.

Referenced Document

This chapter references AN 350: Upgrading Nios Processor Systems to the Nios II Processor.

Document Revision History

Table 15–3 shows the revision history for this chapter.

Table 15–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Chapter 15 was formerly chapter 13. • Added the description for a new parameter, **Enable Bit Clearing for Edge Capture Register**.	—
May 2007 v7.1.0	• Chapter 13 was formerly chapter 11. • Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies • Changed old “Avalon switch fabric” term to “system interconnect fabric” • Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

16. Timer Core

NII51008-7.2.0

Core Overview

The timer core with Avalon® interface is a 32-bit interval timer for Avalon-based processor systems, such as a Nios® II processor system. The timer provides the following features:

- Controls to start, stop, and reset the timer
- Two count modes: count down once and continuous count-down
- Count-down period register
- Maskable interrupt request (IRQ) upon reaching zero
- Optional watchdog timer feature that resets the system if timer ever reaches zero
- Optional periodic pulse generator feature that outputs a pulse when timer reaches zero
- Compatible with 32-bit and 16-bit processors

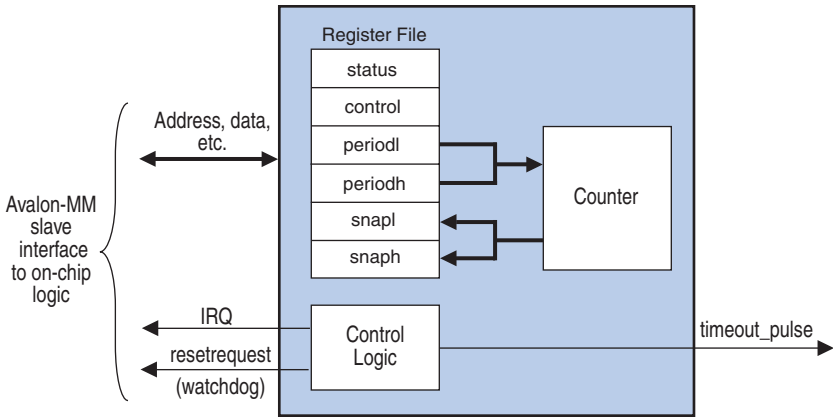
Device drivers are provided in the HAL system library for the Nios II processor. The timer core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 16–2
- "Device and Tools Support" on page 16–3
- "Instantiating the Core in SOPC Builder" on page 16–3
- "Software Programming Model" on page 16–6

Functional Description

Figure 16–1 shows a block diagram of the timer core.

Figure 16–1. Timer Core Block Diagram



The timer core has two user-visible features:

- The Avalon Memory-Mapped (Avalon-MM) interface that provides access to six 16-bit registers
- An optional pulse output that can be used as a periodic pulse generator

All registers are 16-bits wide, making the timer compatible with both 16-bit and 32-bit processors. Certain registers only exist in hardware for a given configuration. For example, if the timer is configured with a fixed period, the period registers do not exist in hardware.

The basic behavior of the timer is described below:

- An Avalon-MM master peripheral, such as a Nios II processor, writes the timer core's `control` register to:
  - Start and stop the timer
  - Enable/disable the IRQ
  - Specify count-down once or continuous count-down mode
- A processor reads the `status` register for information about current timer activity.
- A processor can specify the timer period by writing a value to the period registers, `periodl` and `periodh`.
- An internal counter counts down to zero, and whenever it reaches zero, it is immediately reloaded from the period registers.

- A processor can read the current counter value by first writing to either `snapl` or `snaph` to request a coherent snapshot of the counter, and then reading `snapl` and `snaph` for the full 32-bit value.
- When the count reaches zero:
  - If IRQs are enabled, an IRQ is generated
  - The (optional) pulse-generator output is asserted for one clock period
  - The (optional) watchdog output resets the system

Avalon-MM Slave Interface

The timer core implements a simple Avalon-MM slave interface to provide access to the register file. The Avalon-MM slave port uses the `resetrequest` signal to implement watchdog timer behavior. This signal is a non-maskable reset signal, and it drives the reset input of all Avalon-MM peripherals in the SOPC Builder system. When the `resetrequest` signal is asserted, it forces any processor connected to the system to reboot. For more information, refer to "Configuring the Timer as a Watchdog Timer" on page 16–5.

Device and Tools Support

The timer core supports all Altera® FPGA families.

Instantiating the Core in SOPC Builder

Designers use the MegaWizard® interface for the timer core in SOPC Builder to specify the hardware features. This section describes the options available in the MegaWizard interface.

Timeout Period

The **Timeout Period** setting determines the initial value of the `periodl` and `periodh` registers. When the **Writeable period** setting is enabled, a processor can change the value of the period by writing `periodl` and `periodh`. When **Writeable period** (see below) is off, the period is fixed and cannot be updated at runtime.

The **Timeout Period** is an integer multiple of the **Timer Frequency**. The **Timer Frequency** is fixed at the frequency setting of the system clock associated with the timer. The **Timeout Period** setting can be specified in units of **µs** (microseconds), **ms** (milliseconds), **seconds**, or **clocks** (number of cycles of the system clock associated with the timer). The actual period depends on the frequency of the system clock associated with the timer. If the period is specified in **µs**, **ms**, or **seconds**, the true period will be the smallest number of clock cycles that is greater or equal

to the specified **Timeout Period** value. For example, if the associated system clock has a frequency of 30 ns, and the specified **Timeout Period** value is 1 **µs**, then the true timeout period will be 1.020 microseconds.

Hardware Options

The following options affect the hardware structure of the timer core. As a convenience, the **Preset Configurations** list offers several pre-defined hardware configurations, such as:

- **Simple periodic interrupt**—This configuration is useful for systems that require only a periodic IRQ generator. The period is fixed and the timer cannot be stopped, but the IRQ can be disabled.
- **Full-featured**—This configuration is useful for embedded processor systems that require a timer with variable period that can be started and stopped under processor control.
- **Watchdog**—This configuration is useful for systems that require watchdog timer to reset the system in the event that the system has stopped responding. Refer to "Configuring the Timer as a Watchdog Timer" on page 16–5.

Register Options

Table 16–1 shows the settings that affect the timer core's registers.

Table 16–1. Register Options

Option	Description
Writeable period	When this option is enabled, a master peripheral can change the count-down period by writing `periodl` and `periodh`. When disabled, the count-down period is fixed at the specified **Timeout Period**, and the `periodl` and `periodh` registers do not exist in hardware.
Readable snapshot	When this option is enabled, a master peripheral can read a snapshot of the current count-down. When disabled, the status of the counter is detectable only via other indicators, such as the `status` register or the IRQ signal. In this case, the `snapl` and `snaph` registers do not exist in hardware, and reading these registers produces an undefined value.
Start/Stop control bits	When this option is enabled, a master peripheral can start and stop the timer by writing the START and STOP bits in the `control` register. When disabled, the timer runs continuously. When the **System reset on timeout (watchdog)** option is enabled, the START bit is also present, regardless of the **Start/Stop control bits** option.

Output Signal Options

Table 16–2 shows the settings that affect the timer core's output signals.

Table 16–2. Output Signal Options

Option	Description
Timeout pulse (1 clock wide)	When this option is enabled, the timer core outputs a signal `timeout_pulse`. This signal pulses high for one clock cycle whenever the timer reaches zero. When disabled, the `timeout_pulse` signal does not exist.
System reset on timeout (watchdog)	When this option is enabled, the timer core's Avalon-MM slave port includes the `resetrequest` signal. This signal pulses high for one clock cycle (causing a system-wide reset) whenever the timer reaches zero. When this option is enabled, the internal timer is stopped at reset. Explicitly writing the START bit of the `control` register starts the timer. When this option is disabled, the `resetrequest` signal does not exist. Refer to "Configuring the Timer as a Watchdog Timer".

Configuring the Timer as a Watchdog Timer

To configure the timer for use as a watchdog, in the MegaWizard interface select **Watchdog** in the **Preset Configurations** list, or choose the following settings:

- Set the **Timeout Period** to the desired "watchdog" period.
- Turn off **Writeable period**.
- Turn off **Readable snapshot**.
- Turn off **Start/Stop control bits**.
- Turn off **Timeout pulse**.
- Turn on **System reset on timeout (watchdog)**.

A watchdog timer wakes up (i.e., comes out of reset) stopped. A processor later starts the timer by writing a 1 to the `control` register's START bit. Once started, the timer can never be stopped. If the internal counter ever reaches zero, the watchdog timer resets the system by generating a pulse on its `resetrequest` output. To prevent the system from resetting, the processor must periodically reset the timer's count-down value by writing either the `periodl` or `periodh` registers (the written value is ignored). If the processor fails to access the timer because, for example, software stopped executing normally, then the watchdog timer resets the system and returns the system to a defined state.

Software Programming Model

The following sections describe the software programming model for the timer core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides hardware abstraction layer (HAL) system library drivers that enable you to access the timer core using the HAL application programming interface (API) functions.

HAL System Library Support

The Altera-provided drivers integrate into the HAL system library for Nios II systems. When possible, HAL users should access the timer via the HAL API, rather than accessing the timer registers.

Altera provides a driver for both the HAL timer device models: system clock timer, and timestamp timer.

System Clock Driver

When configured as the system clock, the timer runs continuously in periodic mode, using the default period set in SOPC builder. The system clock services are then run as a part of the interrupt service routine for this timer. The driver is interrupt-driven, and therefore must have its interrupt signal connected in the system hardware.

The Nios II integrated development environment (IDE) allows you to specify system library properties that determine which timer device will be used as the system clock timer.

Timestamp Driver

The timer core may be used as a timestamp device if it meets the following conditions:

- The timer has a writeable `period` register, as configured in SOPC Builder.
- The timer is not selected as the system clock.

The Nios II IDE allows you to specify system library properties that determine which timer device will be used as the timestamp timer.

If the timer hardware is not configured with writeable `period` registers, then calls to the `alt_timestamp_start()` API function will not reset the timestamp counter. All other HAL API calls will perform as expected.

For more information about using the system clock and timestamp features that use these drivers, refer to the *Nios II Software Developer's Handbook*. The Nios II Embedded Design Suite (EDS) also provides several example designs that use the timer core.

Limitations

The HAL driver for the timer core does not support the watchdog reset feature of the timer core.

Software Files

The timer core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_timer_regs.h—**This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_timer.h**, **altera_avalon_timer_sc.c, altera_avalon_timer_ts.c, altera_avalon_timer_vars.c—**These files implement the timer device drivers for the HAL system library.

Register Map

A programmer should never have to directly access the timer via its registers if using the standard features provided in the HAL system library for the Nios II processor. In general, the register map is only useful to programmers writing a device driver.

CAUTION

The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate correctly.

Table 16–3 shows the register map for the timer.

Table 16–3. Register Map (Part 1 of 2)

Offset	Name	R/W	Description of Bits						
			15	…	4	3	2	1	0
0	status	RW	*(1)*					RUN	TO
1	control	RW	*(1)*			STOP	START	CONT	ITO
2	periodl	RW	Timeout Period – 1 (bits 15..0)						
3	periodh	RW	Timeout Period – 1 (bits 31..16)						

Table 16–3. Register Map (Part 2 of 2)

Offset	Name	R/W	Description of Bits						
			15	…	4	3	2	1	0
4	snapl	RW	Counter Snapshot (bits 15..0)						
5	snaph	RW	Counter Snapshot (31..16)						

Note to Table 16–3:
(1) Reserved. Read values are undefined. Write zero.

status Register

The status register has two defined bits, as shown in Table 16–4.

Table 16–4. status Register Bits

Bit	Name	Read/ Write/ Clear	Description
0	TO	RC	The TO (timeout) bit is set to 1 when the internal counter reaches zero. Once set by a timeout event, the TO bit stays set until explicitly cleared by a master peripheral. Write zero to the status register to clear the TO bit.
1	RUN	R	The RUN bit reads as 1 when the internal counter is running; otherwise this bit reads as 0. The RUN bit is not changed by a write operation to the status register.

control Register

The control register has four defined bits, as shown in Table 16–5.

Table 16–5. control Register Bits (Part 1 of 2)

Bit	Name	Read/ Write/ Clear	Description
0	ITO	RW	If the ITO bit is 1, the timer core generates an IRQ when the status register's TO bit is 1. When the ITO bit is 0, the timer does not generate IRQs.
1	CONT	RW	The CONT (continuous) bit determines how the internal counter behaves when it reaches zero. If the CONT bit is 1, the counter runs continuously until it is stopped by the STOP bit. If CONT is 0, the counter stops after it reaches zero. When the counter reaches zero, it reloads with the 32-bit value stored in the periodl and periodh registers, regardless of the CONT bit.

Table 16–5. control Register Bits (Part 2 of 2)

Bit	Name	Read/ Write/ Clear	Description
2	START (1)	W	Writing a 1 to the START bit starts the internal counter running (counting down). The START bit is an event bit that enables the counter when a write operation is performed. If the timer is stopped, writing a 1 to the START bit causes the timer to restart counting from the number currently held in its counter. If the timer is already running, writing a 1 to START has no effect. Writing 0 to the START bit has no effect.
3	STOP *(1)*	W	Writing a 1 to the STOP bit stops the internal counter. The STOP bit is an event bit that causes the counter to stop when a write operation is performed. If the timer is already stopped, writing a 1 to STOP has no effect. Writing a 0 to the stop bit has no effect. Writing 0 to the STOP bit has no effect. If the timer hardware is configured with **Start/Stop control bits** off, writing the STOP bit has no effect.

Note to Table 16–5:
(1) Writing 1 to both START and STOP bits simultaneously produces an undefined result.

periodl and periodh Registers

The `periodl` and `periodh` registers together store the timeout period value. `periodl` holds the least-significant 16 bits, and `periodh` holds the most-significant 16 bits. The internal counter is loaded with the 32-bit value stored in `periodh` and `periodl` whenever one of the following occurs:

- A write operation to either the `periodh` or `periodl` register
- The internal counter reaches 0

The timer's actual period is one cycle greater than the value stored in `periodh` and `periodl`, because the counter assumes the value zero (0×00000000) for one clock cycle.

Writing to either `periodh` or `periodl` stops the internal counter, except when the hardware is configured with **Start/Stop control bits** off. If **Start/Stop control bits** is off, writing either register does not stop the counter. When the hardware is configured with **Writeable period** disabled, writing to either `periodh` or `periodl` causes the counter to reset to the fixed **Timeout Period** specified at system generation time.

snapl and snaph Registers

A master peripheral may request a coherent snapshot of the current 32-bit internal counter by performing a write operation (write-data ignored) to either the `snapl` or `snaph` registers. When a write occurs, the value of the counter is copied to `snapl` and `snaph`. `snapl` holds the least-significant 16 bits of the snapshot and `snaph` holds the most-significant 16 bits. The snapshot occurs whether or not the counter is running. Requesting a snapshot does not change the internal counter's operation.

Interrupt Behavior

The timer core generates an IRQ whenever the internal counter reaches zero and the ITO bit of the `control` register is set to 1. Acknowledge the IRQ in one of two ways:

- Clear the TO bit of the `status` register
- Disable interrupts by clearing the ITO bit of the `control` register

Failure to acknowledge the IRQ produces an undefined result.

Referenced Document

This chapter references the Nios II Software Developer's Handbook.

Document Revision History

Table 16–6 shows the revision history for this chapter.

Table 16–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Chapter 16 was formerly chapter 14. • Updated and expanded definition of Timeout Period	—
May 2007 v7.1.0	• Corrected an error: The timer can be used as a timestamp device if it has a writeable *period* register. • Added table of contents to Overview section. • Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies. Changed old "Avalon switch fabric" term to "system interconnect fabric." Changed old "Avalon interface" terms to "Avalon Memory-Mapped interface." • Added statement that failure to acknowledge an IRQ results in an undefined result in section "Interrupt Behavior" on page 12–9.	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

17. System ID Core

NII51014-7.2.0

Core Overview

The system ID core with Avalon® interface is a simple read-only device that provides SOPC Builder systems with a unique identifier. Nios® II processor systems use the system ID core to verify that an executable program was compiled targeting the actual hardware image configured in the target FPGA. If the expected ID in the executable does not match the system ID core in the FPGA, it is possible that the software will not execute correctly.

This chapter contains the following sections:

- "Functional Description" on page 17–1
- "Device and Tools Support" on page 17–2
- "Instantiating the Core in SOPC Builder" on page 17–2
- "Software Programming Model" on page 17–3

Functional Description

The system ID core provides a read-only Avalon Memory-Mapped (Avalon-MM) slave interface. This interface has two registers, as shown in Table 17–1.

Table 17–1. System ID Core Register Map

Offset	Register Name	R/W	Bit Description
			31...0
0	id	R	SOPC Builder System ID *(1)*
1	timestamp	R	SOPC Builder Generation Time *(1)*

Note to Table 17–1:
(1) Return value is constant.

The value of each register is determined at system generation time, and always returns a constant value. The meaning of the values is:

- id— A unique 32-bit value that is based on the contents of the SOPC Builder system. The id is similar to a check-sum value; SOPC Builder systems with different components, different configuration options, or both, produce different id values.

- timestamp—A unique 32-bit value that is based on the system generation time. The value is equivalent to the number of seconds after Jan. 1, 1970.

There are two basic ways to use the system ID core:

- Verify the system ID before downloading new software to a system. This method is used by software development tools, such as the Nios II integrated development environment (IDE). There is little point in downloading a program to a target hardware system, if the program is compiled for different hardware. Therefore, the Nios II IDE checks that the system ID core in hardware matches the expected system ID of the software before downloading a program to run or debug.
- Check system ID after reset. If a program is running on hardware other than the expected SOPC Builder system, then the program may fail to function altogether. If the program does not crash, it can behave erroneously in subtle ways that are difficult to debug. To protect against this case, a program can compare the expected system ID against the system ID core, and report an error if they do not match.

Device and Tools Support

The system ID core supports all device families supported by SOPC Builder. The system ID core provides a device driver for the Nios II hardware abstraction layer (HAL) system library. No software support is provided for any other processor, including the first-generation Nios processor.

Instantiating the Core in SOPC Builder

The System ID core has no user-configurable features. The id and timestamp register values are determined at system generation time based on the configuration of the SOPC Builder system and the current time. You can add only one system ID core to an SOPC Builder system, and its name is always sysid.

After system generation, you can examine the values stored in the id and timestamp registers by opening the MegaWizard® Plug-In Manager interface for the System ID core. Hovering the mouse over the component in SOPC Builder also displays a tool-tip showing the values.

Software Programming Model

This section describes the software programming model for the system ID core. For Nios II processor users, Altera provides the HAL system library header file that defines the system ID core registers.

The System ID core comes with the following software files. These files provide low-level access to the hardware. Application developers should not modify these files.

- **alt_avalon_sysid_regs.h**—Defines the interface to the hardware registers.
- **alt_avalon_sysid.c, alt_avalon_sysid.h**—Header and source files defining the hardware access functions.

Altera provides one access routine, `alt_avalon_sysid_test()`, that returns a value indicating whether the system ID expected by software matches the system ID core.

alt_avalon_sysid_test()

Prototype:	`alt_32 alt_avalon_sysid_test(void)`
Thread-safe:	No.
Available from ISR:	Yes.
Include:	**<altera_avalon_sysid.h>**
Description:	Returns 0 if the values stored in the hardware registers match the values expected by software. Returns 1 if the hardware timestamp is greater than the software timestamp. Returns -1 if the software timestamp is greater than the hardware timestamp.

Document Revision History

Table 17–2 shows the revision history for this chapter.

Table 17–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 17 was formerly chapter 15.	—
May 2007 v7.1.0	• Chapter 15 was formerly chapter 13. • Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies • Changed old “Avalon switch fabric” term to “system interconnect fabric” • Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
September 2004 v1.1	Updates for Nios II 1.01 release.	—
May 2004 v1.0	Initial release.	—

18. PLL Core

NII53002-7.2.0

Core Overview

The Avalon® memory-mapped (Avalon-MM) phase locked loop (PLL) core with Avalon interface provides a means of accessing the dedicated on-chip PLL circuitry in Altera's Stratix® and Cyclone® series FPGAs. The PLL core is a component wrapper around the Altera® altpll Megafunction.

The core takes an SOPC Builder system clock as its input and generates PLL output clocks locked to that reference clock.

The PLL core supports the following features:

- All PLL features provided by Altera's altpll megafunction. The exact feature set depends on the device family.
- Access to status and control signals via Avalon-MM registers or top-level signals on the SOPC Builder system module.

The PLL output clocks are made available in two ways:

- As sources to system-wide clocks in your SOPC Builder system
- As output signals on your SOPC Builder system module



For details about the altpll megafunction, refer to the *altpll Megafunction User Guide*.

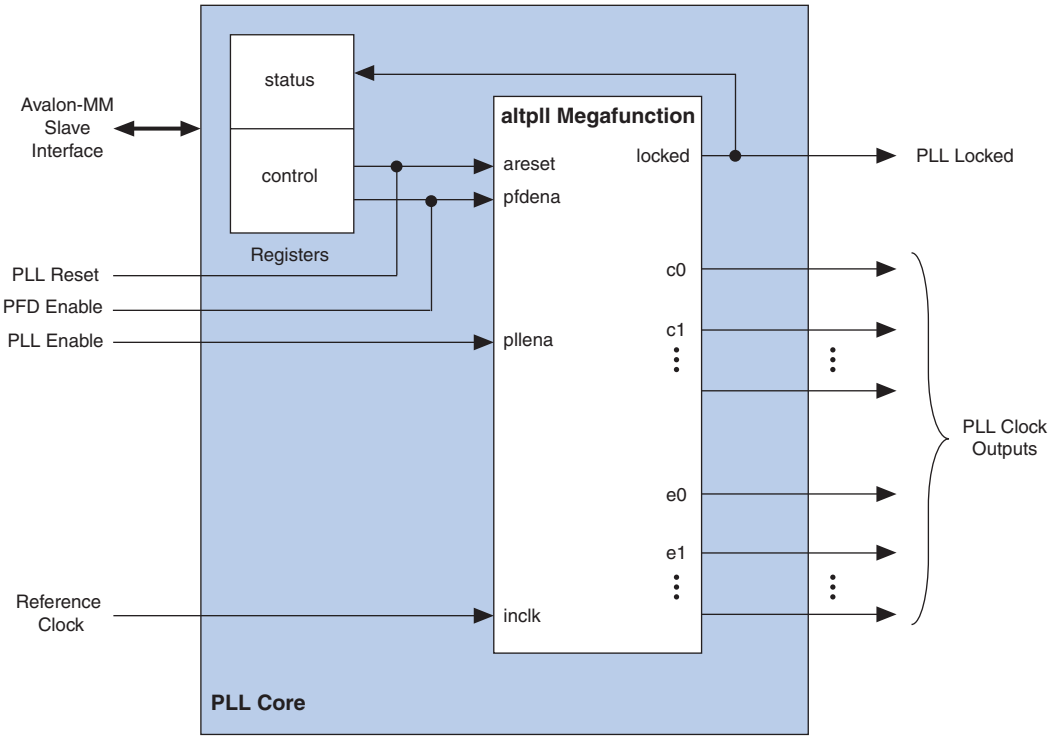
The PLL core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 18–2
- "Device and Tools Support" on page 18–3
- "Instantiating the Core in SOPC Builder" on page 18–4
- "Hardware Simulation Considerations" on page 18–6
- "Register Definitions and Bit List" on page 18–6

Functional Description

Figure 18–1 shows a block diagram of the PLL core and its connection to the PLL circuitry inside an Altera FPGA. The following sections describe the components of the core.

Figure 18–1. PLL Core Block Diagram



altpll Megafunction

The PLL core consists of an altpll megafunction instantiation and an Avalon-MM slave interface. This interface can optionally provide access to status and control registers within the core. The altpll Megafunction takes an SOPC Builder system clock as its reference, and generates one or more phase-locked output clocks.

Clock Outputs

Depending on the target device family, the altpll Megafunction can produce two types of output clock:

- internal (c)—clock outputs that can drive logic either inside or outside the SOPC Builder system module. Internal clock outputs can also be mapped to top-level FPGA pins. Internal clock outputs are available on all device families.
- external (e)—clock outputs that can only drive dedicated FPGA pins. They can not be used as on-chip clock sources. External clock outputs are not available on all device families.



To determine the exact number and type of output clocks available on your target device, refer to the *altpll Megafunction User Guide*.

PLL Status and Control Signals

Depending on how the altpll megafunction is parameterized, there can be a variable number of status and control signals. You can choose to export certain status and control signals to the top-level SOPC Builder system module. Alternatively, Avalon-MM registers can provide access to the signals. Any status or control signals which are not mapped to registers are exported to the top-level module. For details, refer to the "Instantiating the Core in SOPC Builder" on page 18–4.

System Reset Considerations

At FPGA configuration, the PLL core resets automatically. PLL-specific reset circuitry guarantees that the PLL locks before releasing reset for the overall SOPC Builder system module.



Resetting the PLL resets the entire SOPC Builder system module.

Device and Tools Support

The PLL core is supported by the Quartus II software version 5.1 and later. The core supports any Altera FPGA family supported by the altpll megafunction.



For more information about the altpll megafunction, refer to the *altpll Megafunction User Guide*.

Instantiating the Core in SOPC Builder

The PLL core contains an instantiation of the altpll Megafunction. The MegaWizard® interface for the PLL core allows you to configure the altpll, and specify connections to selected altpll status and control signals. The PLL core appears in the **Other** category in the SOPC Builder list of available components.

The following sections describe the options available in the MegaWizard interface for the Avalon-MM PLL core in SOPC Builder.

PLL Settings Page

The **PLL Settings** page contains a button that launches Altera's altpll MegaWizard Plug-In Manager. Use the MegaWizard interface to parameterize the altpll megafunction. The set of available parameters depends on the target device family.



For details about using the altpll MegaWizard interface, refer to the *altpll Megafunction User Guide*.

You cannot click **Finish** in the Avalon-MM PLL wizard nor configure the PLL interface until you parameterize the altpll megafunction.

Interface Page

The **Interface** page configures the access modes for the optional advanced PLL status and control signals.

For each advanced signal present on the altpll, you can select one of the following access modes:

- **Export**—Exports the signal to the top level of the SOPC builder system module.
- **Register**—Maps the signal to a bit in a status or control register.



The advanced signals are optional. If you choose not to create any of them in the altpll MegaWizard, the PLL's default behavior will be as shown in Table 18–1.

You can specify the access mode for the advanced signals shown in Table 18–1. The altpll core signals, not displayed in this table, are automatically exported to the top level of the SOPC Builder system module.

Table 18–1. altpll Advanced Signals

altpll Name	Input / Output	Avalon-MM PLL Wizard Name	Default Behavior	Description
areset	input	PLL Reset Input	The PLL is reset only at device configuration.	This signal resets the entire SOPC Builder system module, and restores the PLL to its initial settings.
pllena	input	PLL Enable Input	The PLL is enabled.	This signal enables the PLL. pllena is always exported.
pfdena	input	PFD Enable Input	The phase-frequency detector is enabled.	This signal enables the phase-frequency detector in the PLL, allowing it to lock on to changes in the clock reference.
locked	output	PLL Locked Output	—	This signal is asserted when the PLL is locked to the input clock.



Asserting areset resets the entire SOPC Builder system module, not just the PLL.

Finish

Click **Finish** to insert the PLL into the SOPC Builder system. The PLL clock output(s) appear in the clock settings table on the SOPC Builder **System Contents** tab.



If the PLL has external output clocks, they appear in the clock settings table like other clocks; however, you cannot use them to drive components within the SOPC Builder system.



For details about using external output clocks, refer to the *altpll Megafunction User Guide*.

The SOPC Builder automatically connects the PLL's reference clock input to the first available clock in the clock settings table.



If there is more than one SOPC Builder system clock available, verify that the PLL is connected to the appropriate reference clock.

Hardware Simulation Considerations

The HDL files generated by SOPC Builder for the PLL core are suitable for both synthesis and simulation. The PLL core supports the standard SOPC Builder simulation flow, so there are no special considerations for hardware simulation.

Register Definitions and Bit List

Table 18–2 shows the register map for the PLL core. Device drivers can control and communicate with the core through two 16-bit memory-mapped registers, `status` and `control`.

Note that the status and control bits shown below are present only if they have been created in the altpll MegaWizard, and set to **Register** on the **Interface** page in the PLL wizard.

Table 18–2. PLL Core Register Map

Offset	Register Name	R/W	Bit Description				
			15	…	2	1	0
0	`status`	R/O	*(1)*				LOCKED
1	`control`	R/W	*(1)*			PFDENA	ARESET

Note to Table 18–2:
(1) Reserved. Read values are undefined. When writing, set reserved bits to zero.

Status Register

Embedded software can access the PLL status via the `status` register. Writing to `status` has no effect. Table 18–3 describes the function of each bit.

Table 18–3. Status Register Bits

Bit Number	Bit Name	Value after reset	Description
0	LOCKED	1	Connects to the `locked` signal on the altpll. The `LOCKED` bit is high when valid clocks are present on the output of the PLL.
1 .. 15	—	—	Reserved. Read values are undefined.

Control Register

Embedded software can control the PLL via the `control` register. Software can also read back the status of control bits. Table 18–4 describes the function of each bit.

Table 18–4. Control Register Bits

Bit Number	Bit Name	Value after reset	Description
0	ARESET	0	Connects to the `areset` signal on the altpll. Writing a 1 to this bit asserts the `areset` signal for one clock cycle.
1	PFDENA	1	Connects to the `pfdena` signal on the altpll.
2 .. 15	—	—	Reserved. Read values are undefined. When writing, set reserved bits to zero.

Referenced Document

This chapter references the *altpll Megafunction User Guide*.

Document Revision History

Table 18–5 shows the revision history for this chapter.

Table 18–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 18 was formerly chapter 16.	—
May 2007 v7.1.0	• Chapter 16 was formerly chapter 14. • Added table of contents to Overview section. • Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies • Changed old “Avalon switch fabric” term to “system interconnect fabric” • Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	Initial release.	—

19. Performance Counter Core

QIII55001-7.2.0

Core Overview

The performance counter core with Avalon® interface enables relatively unobtrusive, real-time profiling of software programs. With the performance counter, you can accurately measure execution time taken by multiple sections of code. You need only add a single instruction at the beginning and end of each section to be measured.

The main benefit of using the performance counter core is the accuracy of the profiling results. Alternatives include the following approaches:

- GNU profiler, `gprof`—`gprof` provides broad low-precision timing information about the entire software system. It uses a substantial amount of RAM, and degrades the real-time performance. For many embedded applications, `gprof` distorts real-time behavior too much to be useful.
- Interval timer peripheral—The interval timer is less intrusive than `gprof`. It can provide good results for narrowly targeted sections of code.

The performance counter core is unobtrusive, requiring only a single instruction to start and stop profiling, and no RAM. It is appropriate for high-precision measurements of narrowly targeted sections of code.



For further discussion of all three profiling methods, refer to *AN 391: Profiling Nios II Systems*.

The performance counter core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. The core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera® provides device drivers to enable the Nios II processor to use the performance counters.

This chapter contains the following sections:

- "Functional Description" on page 19–2
- "Device and Tools Support" on page 19–4
- "Instantiating the Core in SOPC Builder" on page 19–4
- "Hardware Simulation Considerations" on page 19–4
- "Software Programming Model" on page 19–5
- "Performance Counter API" on page 19–8

Functional Description

The performance counter core is a set of counters which track clock cycles, timing multiple sections of your software. You can start and stop these counters in your software, individually or as a group. You can read cycle counts from hardware registers.

The core contains two counters for every section:

- Time: A 64-bit clock cycle counter
- Events: A 32-bit event counter

Section Counters

Each 64-bit time counter records the aggregate number of clock cycles spent in a section of code. The 32-bit event counter records the number of times the section executes.

The performance counter core can have up to seven section counters.

Global Counter

The global counter controls all section counters. The section counters are enabled only when the global counter is running.

The 64-bit global clock cycle counter tracks the aggregate time for which the counters were enabled. The 32-bit global event counter tracks the number of global events, that is, the number of times the performance counter core has been enabled.

Register Map

The performance counter core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to memory-mapped registers. Reading from the registers retrieves the current times and event counts. Writing to the registers starts, stops and resets the counters. Table 19–1 shows the registers in detail.

Table 19–1. Performance Counter Core Register Map

Offset	Register Name	Bit Description		
		Read	Write	
		31 … 0	31 … 1	0
0	T[0]$_{lo}$	global clock cycle counter [31: 0]	*(1)*	0 = STOP 1 = RESET
1	T[0]$_{hi}$	global clock cycle counter [63:32]	*(1)*	0 = START
2	Ev[0]	global event counter	*(1)*	*(1)*
3	—	*(1)*	*(1)*	*(1)*
4	T[1]$_{lo}$	section 1 clock cycle counter [31: 0]	*(1)*	0 = STOP
5	T[1]$_{hi}$	section 1 clock cycle counter [63:32]	*(1)*	0 = START
6	Ev[1]	section 1 event counter	*(1)*	*(1)*
7	—	*(1)*	*(1)*	*(1)*
8	T[2]$_{lo}$	section 2 clock cycle counter [31: 0]	*(1)*	0 = STOP
9	T[2]$_{hi}$	section 2 clock cycle counter [63:32]	*(1)*	0 = START
10	Ev[2]	section 2 event counter	*(1)*	*(1)*
11	—	*(1)*	*(1)*	*(1)*
. . .	. . .	. . .	. . .	. . .
4*n* + 0	T[n]$_{lo}$	section *n* clock cycle counter [31: 0]	*(1)*	0 = STOP
4*n* + 1	T[n]$_{hi}$	section *n* clock cycle counter [63:32]	*(1)*	0 = START
4*n* + 2	Ev[n]	section *n* event counter	*(1)*	*(1)*
4*n* + 3	—	*(1)*	*(1)*	*(1)*

Note to Table 19–1:

(1) Reserved. Read values are undefined. When writing, set reserved bits to zero.

System Reset Considerations

After system reset, the performance counter core is stopped and disabled, and all counters contain zero.

Device and Tools Support

The performance counter core supports all Altera device families supported by SOPC Builder, and provides device drivers for the Nios II hardware abstraction layer (HAL) system library.

Instantiating the Core in SOPC Builder

Designers use the MegaWizard® interface for the performance counter core in SOPC Builder to specify the core's hardware features.

Define Counters

Choose the number of section counters you want to generate by selecting from the "**Number of simultaneously-measured sections"** list. The performance counter core may have up to seven sections. If you require more that seven sections, you can instantiate multiple performance counter cores.

Multiple Clock Domain Considerations

If your SOPC Builder system uses multiple clocks, place the performance counter core in the same clock domain as the CPU. Otherwise, it is not possible to convert cycle counts to seconds correctly.

Hardware Simulation Considerations

You can use this core in simulation with no special considerations.

Software Programming Model

The following sections describe the software programming model for the performance counter core.

Software Files

Altera provides the following software files for Nios II systems. These files define the low-level access to the hardware and provide control and reporting functions. Do not modify these files.

- **altera_avalon_performance_counter.h, altera_avalon_performance_counter.c** —The header and source code for the functions and macros needed to control the performance counter core and retrieve raw results.
- **perf_print_formatted_report.c**—The source code for simple profile reporting.

Using the Performance Counter

In a Nios II system, you can control the performance counter core with a set of highly efficient C macros, and extract the results with C functions.

API Summary

The Nios II application program interface (API) for the performance counter core consists of functions, macros and constants.

Functions and macros

Table 19–2 lists macros and functions for accessing the performance counter hardware structure.

Table 19–2. Performance Counter Macros and Functions

Name	Summary
`PERF_RESET()`	Stops and disables all counters, resetting them to 0.
`PERF_START_MEASURING()`	Starts the global counter and enables section counters.
`PERF_STOP_MEASURING()`	Stops the global counter and disables section counters.
`PERF_BEGIN()`	Starts timing a code section.
`PERF_END()`	Stops timing a code section.
`perf_print_formatted_report()`	Sends a formatted summary of the profiling results to stdout.
`perf_get_total_time()`	Returns the aggregate global profiling time in clock cycles.
`perf_get_section_time()`	Returns the aggregate time for one section in clock cycles.

Table 19–2. Performance Counter Macros and Functions

Name	Summary
`perf_get_num_starts()`	Returns the number of counter events.
`alt_get_cpu_freq()`	Returns the CPU frequency in Hz.

For a complete description of each macro and function, see "Performance Counter API" on page 19–8.

Hardware constants

You can get the performance counter hardware parameters from constants defined in **system.h**. The constant names are based on the performance counter instance name, specified on the **System Contents** tab in SOPC Builder. Table 19–3 lists the hardware constants.

Table 19–3. Performance Counter Constants

Name *(1)*	Meaning
`PERFORMANCE_COUNTER_BASE`	Base address of core
`PERFORMANCE_COUNTER_SPAN`	Number of hardware registers
`PERFORMANCE_COUNTER_HOW_MANY_SECTIONS`	Number of section counters

Note to Table 19–3:
(1) Example based on instance name `performance_counter`

Startup

Before using the performance counter core, invoke `PERF_RESET` to stop, disable and zero all counters.

Global Counter Usage

Use the global counter to enable and disable the entire performance counter core. For example, you might choose to leave profiling disabled until your software has completed its initialization.

Section Counter Usage

To measure a section in your code, surround it with the macros `PERF_BEGIN()` and `PERF_END()`. These macros consist of a single write to the performance counter core.

You can simultaneously measure as many code sections as you like, up to the number specified in SOPC Builder. See "Define Counters" on page 19–4 for details. You can start and stop counters individually, or as a group.

Typically, you assign one counter to each section of code you intend to profile. However, in some situation you may wish to group several sections of code in a single section counter. As an example, to measure general interrupt overhead, you can measure all interrupt service routines (ISRs) with one counter.

To avoid confusion, assign a mnemonic symbol for each section number.



For an example, refer to the performance checksum design files accompanying *AN 391: Profiling Nios II Systems*. These files may be found on the Altera Nios II literature page at www.altera.com/literature/lit-nio2.jsp.

Viewing Counter Values

Library routines allow you to retrieve and analyze the results. Use `perf_print_formatted_report()` to list the results to `stdout` as shown in Example 19–1.

Example 19–1.

```
perf_print_formatted_report(
    (void *)PERFORMANCE_COUNTER_BASE,  // Peripheral's HW base address
    alt_get_cpu_freq(),                // defined in "system.h"
    3,                                 // How many sections to print
    "1st checksum_test",               // Display-names of sections
    "pc_overhead",
   "ts_overhead");
```

Example 19–2 creates a table similar to this result.

Example 19–2.

```
--Performance Counter Report--
Total Time: 2.07711 seconds (103855534 clock-cycles)
+------------------+--------+-----------+----------------+-----------+
| Section          |      % | Time (sec)| Time (clocks) |Occurrences|
+------------------+--------+-----------+----------------+-----------+
|1st checksum_test|     50 |   1.03800 |      51899750 |         1 |
+------------------+--------+-----------+----------------+-----------+
| pc_overhead     |1.73e-05|   0.00000 |             18 |         1 |
+------------------+--------+-----------+----------------+-----------+
| ts_overhead     |4.24e-05|   0.00000 |             44 |         1 |
+----------------+-------+----------+--------------+----------+
```

For full documentation of `perf_print_formatted_report()`, see "Performance Counter API" on page 19–8.

Interrupt Behavior

The performance counter core does not generate interrupts.

You can start and stop performance counters, and read raw performance results, in an interrupt service routine (ISR). Do not call function `perf_print_formatted_report()` from an ISR.



If a interrupt occurs during the measurement of a section of code, the time taken by the CPU to process the interrupt and return to the section is added to the measurement time. The same applies to context switches in a multithreaded environment. Your software must take appropriate measures to avoid or handle these situations.

Performance Counter API

This section describes the application programming interface (API) for the performance counter core.

For Nios II processor users, Altera provides routines to access the performance counter core hardware. These functions are specific to the performance counter core and directly manipulate low level hardware. The performance counter core cannot be accessed via the HAL API or the ANSI C standard library.

PERF_RESET()

Prototype:	`PERF_RESET(p)`
Thread-safe:	Yes
Available from ISR:	Yes
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`p`—performance counter core base address
Returns:	—
Description:	Macro `PERF_RESET()` stops and disables all counters, resetting them to 0.

PERF_START_MEASURING()

Prototype:	`PERF_START_MEASURING(p)`
Thread-safe:	Yes
Available from ISR:	Yes
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`p`—performance counter core base address
Returns:	—
Description:	Macro `PERF_START_MEASURING()` starts the global counter, enabling the performance counter core. The behavior of individual section counters is controlled by `PERF_BEGIN()` and `PERF_END()`. `PERF_START_MEASURING()` defines the start of a global event, and increments the global event counter. This macro is a single write to the performance counter core.

PERF_STOP_MEASURING()

Prototype:	`PERF_STOP_MEASURING(p)`
Thread-safe:	Yes
Available from ISR:	Yes
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`p`—performance counter core base address
Returns:	—
Description:	Macro `PERF_STOP_MEASURING()` stops the global counter, disabling the performance counter core. This macro is a single write to the performance counter core.

PERF_BEGIN()

Prototype:	`PERF_BEGIN(p,n)`
Thread-safe:	Yes
Available from ISR:	Yes
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`p`—performance counter core base address `n`—counter section number. Section counter numbers start at 1. Do not refer to counter 0 in this macro.
Returns:	—
Description:	Macro `PERF_BEGIN()` starts the timer for a code section, defining the beginning of a section event, and incrementing the section event counter. If you subsequently use PERF_STOP_MEASURING() and PERF_START_MEASURING() to disable and re-enable the core, the section counter will resume. This macro is a single write to the performance counter core.

PERF_END()

Prototype:	`PERF_END(p,n)`
Thread-safe:	Yes
Available from ISR:	Yes
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`p`—performance counter core base address `n`—counter section number. Section counter numbers start at 1. Do not refer to counter 0 in this macro.
Returns:	—
Description:	Macro `PERF_END()` stops timing a code section. The section counter does not run, regardless whether the core is enabled or not. This macro is a single write to the performance counter core.

perf_print_formatted_report()

Prototype:

```
int perf_print_formatted_report (
    void* perf_base,
    alt_u32 clock_freq_hertz,
    int num_sections, ...)
```

Thread-safe: No

Available from ISR: No

Include: **<altera_avalon_performance_counter.h>**

Parameters:

`perf_base`—performance counter core base address
`clock_freq_hertz`—clock frequency
`num_sections`—The number of section counters to display. This must not exceed <*instance_name*>`_HOW_MANY_SECTIONS`.

Returns: 0

Description:

Function `perf_print_formatted_report()` reads the profiling results from the performance counter core, and prints a formatted summary table
This function disables all counters. However, for predictable results in a multi-threaded or interrupt environment, invoke `PERF_STOP_MEASURING()` when you reach the end of the code to be measured, rather than relying on `perf_print_formatted_report()`.



This function requires the C standard library. Do not use the small C library with this function.

perf_get_total_time()

Prototype:	`alt_u64 perf_get_total_time(void* hw_base_address)`
Thread-safe:	No
Available from ISR:	Yes
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`hw_base_address`—base address of performance counter core
Returns:	Aggregate global time in clock cycles
Description:	Function `perf_get_total_time()` reads the raw global time. This is the aggregate time, in clock cycles, that the performance counter core has been enabled. This function has the side effect of stopping the counters.

perf_get_section_time()

Prototype:	`alt_u64 perf_get_section_time (void* hw_base_address, int which_section)`
Thread-safe:	No
Available from ISR:	Yes
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`hw_base_address`—performance counter core base address which_section—counter section number
Returns:	Aggregate section time in clock cycles
Description:	Function `perf_get_section_time()` reads the raw time for a given section. This is the time, in clock cycles, that the section has been running. This function has the side effect of stopping the counters.

perf_get_num_starts()

Prototype:
```
alt_u32 perf_get_num_starts
    (void* hw_base_address, int which_section)
```

Thread-safe: Yes

Available from ISR: Yes

Include: **<altera_avalon_performance_counter.h>**

Parameters:
`hw_base_address`—performance counter core base address
`which_section`—counter section number

Returns: Number of counter events

Description: Function `perf_get_num_starts()` retrieves the number of counter events (or times a counter has been started). If which_section = 0, it retrieves the number of global events (times the performance counter core has been enabled). This function does not stop the counters.

alt_get_cpu_freq()

Prototype:	`alt_u32 alt_get_cpu_freq()`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	
Returns:	CPU frequency in Hz
Description:	Function `alt_get_cpu_freq()` returns the CPU frequency in Hz.

Referenced Document

This chapter references *AN 391: Profiling Nios II Systems*.

Document Revision History

Table 19–4 shows the revision history for this chapter.

Table 19–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	• Chapter 19 was formerly chapter 17 • Removed incorrect statement about granularity of the timer.	—
May 2007 v7.1.0	• Chapter 17 was formerly chapter 15. • Added table of contents to Overview section. • Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	• Updated Avalon terminology because of changes to Avalon technologies • Changed old “Avalon switch fabric” term to “system interconnect fabric” • Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface”	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some re-phrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
December 2005 v5.1.0	Initial release.	—

Section VI. Streaming Peripherals

This section describes streaming peripherals provided by Altera® for SOPC Builder systems. These components allow you to optimize streaming applications.

Refer to *About This Handbook* for further details.

This section includes the following chapter:

- Chapter 20, Avalon Streaming Channel Multiplexer and Demultiplexer Cores
- Chapter 21, Avalon Streaming Test Pattern Generator and Checker Cores



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

20. Avalon Streaming Channel Multiplexer and Demultiplexer Cores

QII55004-7.2.0

Core Overview

The Avalon® streaming (Avalon-ST) channel multiplexer receives data from a number of input interfaces and multiplexes the data into a single output interface, using the optional `channel` signal to indicate which input the output data is from. The Avalon-ST channel demultiplexer receives data from a channelized input interface and drives that data to multiple output interfaces, where the output interface is selected by the input `channel` signal.

The multiplexer and demultiplexer can transfer data between interfaces on cores that support the unidirectional flow of data. The multiplexer and demultiplexer allow you to create multiplexed or demultiplexed datapaths without having to write custom HDL code to perform these functions. The multiplexer includes a round-robin scheduler. Both cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Multiplexer" on page 20–3
- "Demultiplexer" on page 20–6
- "Device and Tools Support" on page 20–8
- "Installation and Licensing" on page 20–9
- "Hardware Simulation Considerations" on page 20–9
- "Software Programming Model" on page 20–9

Resource Usage and Performance

Resource utilization for the cores depends upon the number of input and output interfaces, the width of the datapath and whether the streaming data uses the optional packet protocol. For the multiplexer, the parameterization of the scheduler also effects resource utilization. Table 20–1 provides estimated resource utilization for eleven different configurations of the multiplexer.

Table 20–1. Multiplexer Estimated Resource Usage and Performance

No. of Inputs	Data Width	Scheduling Size (Cycles)	Stratix® II and Stratix II GX (Approximate LEs)		Cyclone® II		Stratix	
			f_{MAX} (MHz)	ALM Count	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells
2	Y	1	500	31	420	63	422	80
2	Y	2	500	36	417	60	422	58
2	Y	32	451	43	364	68	360	49
8	Y	2	401	150	257	233	228	298
8	Y	32	356	151	219	207	211	123
16	Y	2	262	333	174	533	170	284
16	Y	32	310	337	161	471	157	277
2	N	1	500	23	400	48	422	52
2	N	9	500	30	420	52	422	56
11	N	9	292	275	197	397	182	287
16	N	9	262	295	182	441	179	224

Table 20–2 provides estimated resource utilization for six different configurations of the demultiplexer. The core operating frequency varies with the device, the number of interfaces and the size of the datapath.

Table 20–2. Demultiplexer Estimated Resource Usage

No. of Inputs	Data Width (Symbols per Beat)	Stratix II (Approximate LEs)		Cyclone II		Stratix II GX (Approximate LEs)	
		f_{MAX} (MHz)	ALM Count	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells
2	1	500	53	400	61	399	44
15	1	349	171	235	296	227	273
16	1	363	171	233	294	231	290
2	2	500	85	392	97	381	71
15	2	352	247	213	450	210	417
16	2	328	280	218	451	222	443

Multiplexer

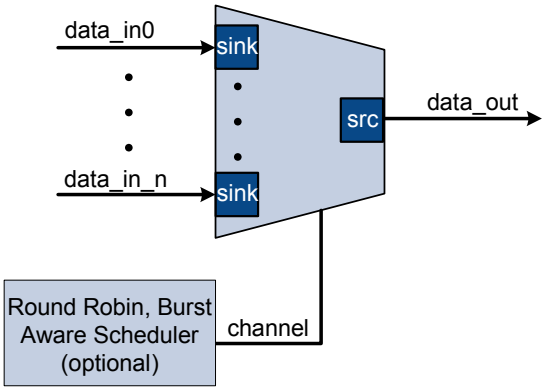
This section describes the hardware structure and functionality of the multiplexer component.

Functional Description

The Avalon-ST multiplexer takes data from a number of input data interfaces, and multiplexes the data onto a single output interface. The mux includes a simple, round-robin scheduler that selects from the next input interface that has data. Each input interface has the same width as the output interface, so that all other input interfaces are backpressured when the mux is carrying data from a different input interface.

The mux includes an optional `channel` signal that enables each input interface to carry channelized data. When the `channel` signal is present on input interfaces, the mux adds $\log_2$(`num_input_interfaces`) bits to make the output channel signal, such that the output channel signal has all of the bits of the input channel plus the bits required to indicate which input interface each cycle of data is from. These bits are appended to either the most or least significant bits of the output `channel` signal as specified in the SOPC Builder MegaWizard® interface (Figure 20–1).

Figure 20–1. Multiplexer



The internal scheduler considers one input interface at a time, selecting it for transfer. Once an input interface has been selected, data from that input interface is sent until one of the following scenarios occurs:

- The specified number of cycles have elapsed
- The input interface has no more data to send and de-asserts `valid` on a ready cycle
- The packets are supported, `endofpacket` is asserted

Input Interfaces

Each input interface is an Avalon-ST data interface that optionally supports packets. The input interfaces are identical; they have the same symbol and data widths, error widths, and channel widths.

Output Interface

The output interface carries the multiplexed data stream with data from all of the inputs. The symbol, data, and error widths are the same as the input interfaces. The width of the `channel` signal is the same as the input interfaces, with the addition of the bits needed to indicate the input each datum was from.

Instantiating the Multiplexer in SOPC Builder

Use the MegaWizard interface for the multiplexer core in SOPC Builder to specify the core configuration. The following sections list the available options in the MegaWizard interface.

Functional Parameters—The following sections outline the options for the multiplexer as a whole:

- **Number of Input Ports**—The number of input interfaces that the multiplexer supports. Valid values are 2 .. 16.
- **Scheduling Size (Cycles)**—The number of cycles that are sent from a single channel before changing to the next channel.
- **Use high bits to indicate source port**—When selected, the high bits of the output channel signal are used to indicate the input interface that the data came from. For example, if the input interfaces have 4-bit channel signals, and the mux has 4 input interfaces, then the output interface has a 6-bit channel signal. If this parameter is true, bits [5:4] of the output channel signal indicate the input interface the data is from, and bits [3:0] are the channel bits that were presented at the input interface.

Output Interface—The following sections outline the options for the output interface:

- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1 – 32 bits.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat (transfer). Valid values are 1 – 32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**—The number of bits used for the `channel` signal for input interfaces. A value of 0 indicates that input interfaces do not have channels. A value of 4 indicates that up to 16 channels share the same input interface. The input channel can have a width between 0-31 bits. A value of 0 means that the optional `channel` signal is not used.
- **Error Signal Width (bits)**—The width of the `error` signal for input and output interfaces. A value of 0 means the `error` signal is not used.

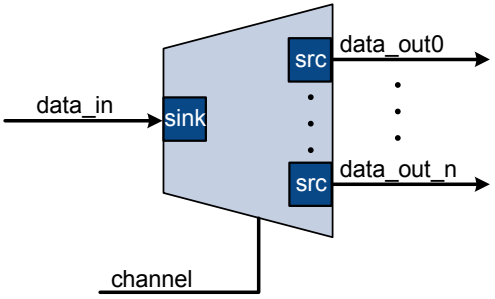
Demultiplexer

This section describes the hardware structure and functionality of the demultiplexer component.

Functional Description

That Avalon-ST demultiplexer takes data from a channelized input data interface and provides that data to multiple output interfaces, where the output interface selected for a particular transfer is specified by the input `channel` signal. The data is delivered to the output interfaces in the same order it was received at the input interface, regardless of the value of `channel`, `packet`, `frame`, or any other signal. Each of the output interfaces has the same width as the input interface, so that each output interface will be idle when the demux is driving data to a different output interface. The demux uses $\log_2$(`num_output_interfaces`) bits of the `channel` signal to select the output to which to forward the data; the remainder of the channel bits are forwarded to the appropriate output interface unchanged (Figure 20–2).

Figure 20–2. The Demultiplexer



Input Interface

Each input interface is an Avalon-ST data interface that optionally supports packets.

Output Interfaces

Each output interface carries data from a subset of channels from the input interface. Each output interface is identical; all have the same symbol and data widths, error widths, and channel widths. The symbol, data, and error widths are the same as the input interface. The width of the `channel` signal is the same as the input interface, without the bits that were used to select the output interface.

Instantiating the Demultiplexer in SOPC Builder

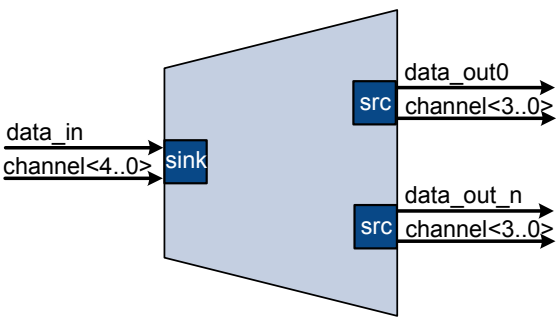
Use the MegaWizard interface for the demultiplexer core in SOPC Builder to specify the core configuration. The following sections list the available options in the MegaWizard interface.

Functional Parameters—The following sections outline the options for the demultiplexer as a whole:

- **Number of Output Ports**—The number of output interfaces that the multiplexer supports Valid values are 2 .. 16.
- **High channel bits select output**—When selected, the high bits of the input channel signal are used by the de-multiplexing function and the low order bits are passed to the output. When not selected, the low order bits are used and the high order bits are passed through.

  The following example illustrates the significance of the location of these signals. In Figure 20–3 there is one input interface and two output interfaces. If the low-order bits of the channel signal select the output interfaces, the even channels will go to channel 0 and the odd channels will go to channel 1. If the high-order bits of the channel signal select the output interface, channels 0–7 will go to channel 0 and channels 8–15 will go to channel 1.

Figure 20–3. Select Bits for Demultiplexer



Input Interface—The following sections outline the options for the input interface.

- **Data Bits Per Symbol -** The number of bits per symbol for the input and output interfaces. Valid values are 1 – 32 bits.
- **Data Symbols Per Beat** - The number of symbols (words) that are transferred per beat (transfer). Valid values are 1 – 32.

- **Include Packet Support** - Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**- The number of bits used for the `channel` signal for output interfaces. A value of 0 means that output interfaces do not use the optional `channel` signal.
- **Error Signal Width (bits)** - The width of the `error` signal for input and output interfaces. A value of 0 means the `error` signal is not unused.

Device and Tools Support

Altera device support for the multiplexer and demultiplexer components is listed in Table 20–3. For each device family, a component provides either full or preliminary support:

- *Full support* means the component meets all functional and timing requirements for the device family and may be used in production designs.
- *Preliminary support* means the component meets all functional requirements, but might still be undergoing timing analysis for the device family; it may be used in production designs with caution.

Table 20–3. Device Family Support

Device Family	Avalon-ST Multiplexer	Avalon-ST Demultiplexer
Arria™ GX	Preliminary	Preliminary
Cyclone III	Preliminary	Preliminary
Cyclone II	Full	Full
Cyclone	Full	Full
HardCopy® II	Full	Full
Stratix III	Preliminary	Preliminary
Stratix II GX	Full	Full
Stratix II	Full	Full
Stratix GX	Full	Full
Stratix	Full	Full

Installation and Licensing

The multiplexer and demultiplexer components are included in the Altera MegaCore® IP Library, which is an optional part of the Quartus® II software installation. After you install the MegaCore IP Library, SOPC Builder recognizes these components and can instantiate them into a system.

You can use the multiplexer and demultiplexer components for free without a license in any design targeting an Altera device.

Hardware Simulation Considerations

The multiplexer and demultiplexer components do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

The multiplexer and demultiplexer components do not have any user-visible control or status registers. Therefore software cannot control or configure any aspect of the multiplexer or demultiplexer at run-time. The components cannot generate interrupts.

Document Revision History

Table 20–4 shows the revision history for this chapter.

Table 20–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Chapter 20 was formerly chapter 18.	—
May 2007 v7.1.0	Initial release.	—

21. Avalon Streaming Test Pattern Generator and Checker Cores

QII55007-7.2.0

Core Overview

The data generation and monitoring solution for Avalon® Streaming (Avalon-ST) consists of two components: a test pattern generator core that generates packetized or non-packetized data and sends it out on an Avalon-ST data interface, and a test pattern checker core that receives the same data and checks it for correctness.

The test pattern generator core can insert different error conditions, and the test pattern checker reports these error conditions to the control interface, each via an Avalon Memory-Mapped (Avalon-MM) slave.

Both cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system.

This chapter contains the following sections:

- "Resource Utilization and Performance" on page 21–2
- "Test Pattern Generator" on page 21–4
- "Test Pattern Checker" on page 21–6
- "Device and Tools Support" on page 21–8
- "Installation and Licensing" on page 21–9
- "Hardware Simulation Considerations" on page 21–9
- "Software Programming Model" on page 21–10
- "Test Pattern Generator API" on page 21–15
- "Test Pattern Checker API" on page 21–21

Resource Utilization and Performance

Resource utilization and performance for the test pattern generator and checker cores depend on the datawidth, number of channels, and whether the streaming data uses the optional packet protocol.

Table 21–1 provides estimated resource utilization and performance for the test pattern generator core.

Table 21–1. Test Pattern Generator Estimated Resource Utilization and Performance

No. of Channels	Datawidth (No. of 8-bit Symbols Per Beat)	Packet Support	Stratix® II and Stratix II GX			Cyclone® II			Stratix		
			f_{MAX} (MHz)	ALM Count	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)
1	4	Yes	284	233	560	206	642	560	202	642	560
1	4	No	293	222	496	207	572	496	245	561	496
32	4	Yes	276	270	912	210	683	912	197	707	912
32	4	No	323	227	848	234	585	848	220	630	848
1	16	Yes	298	361	560	228	867	560	245	896	560
1	16	No	340	330	496	230	810	496	228	845	496
32	16	Yes	295	410	912	209	954	912	224	956	912
32	16	No	269	409	848	219	842	848	204	912	848

Table 21–2 provides estimated resource utilization and performance for the test pattern checker core.

Table 21–2. Test Pattern Checker Estimated Resource Utilization and Performance

No. of Channels	Datawidth (No. of 8-bit Symbols Per Beat)	Packet Support	Stratix®II and Stratix II GX			Cyclone® II			Stratix		
			f_{MAX} (MHz)	ALM Count	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)
1	4	Yes	270	271	96	179	940	0	174	744	96
1	4	No	371	187	32	227	628	0	229	663	32
32	4	Yes	185	396	3616	111	875	3854	105	795	3616
32	4	No	221	363	3520	133	686	3520	133	660	3520
1	16	Yes	253	462	96	185	1433	0	166	1323	96
1	16	No	277	306	32	218	1044	0	192	1004	32
32	16	Yes	182	582	3616	111	1367	3584	110	1298	3616
32	16	No	218	473	3520	129	1143	3520	126	1074	3520

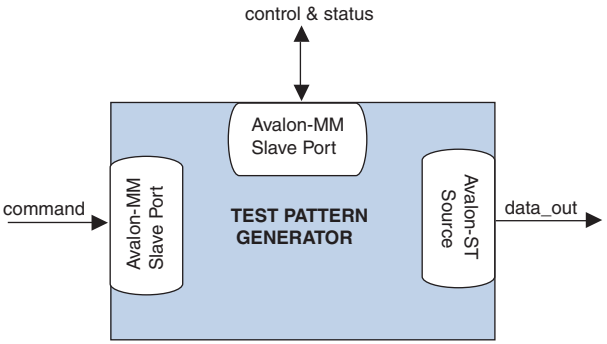
Test Pattern Generator

This section describes the hardware structure and functionality of the test pattern generator core.

Functional Description

The test pattern generator core accepts commands to generate data via an Avalon-MM command interface, and drives the generated data to an Avalon-ST data interface. You can parameterize most aspects of the Avalon-ST data interface such as the number of error bits and data signal width, thus allowing you to test components with different interfaces. Figure 21–1 shows a block diagram of the test pattern generator core.

Figure 21–1. Test Pattern Generator Core Block Diagram



The data pattern is determined by the following equation:
Symbol Value = Symbol Position in Packet XOR Data Error Mask.
Non-packetized data is one long stream with no beginning or end.

The test pattern generator core has a throttle register that is set via the Avalon-MM control interface. The value of the throttle register is used in conjunction with a pseudo-random number generator to throttle the data generation rate.

Command Interface

The command interface is a 32-bit Avalon-MM write slave that accepts data generation commands. It is connected to a 16-element deep FIFO, thus allowing a master peripheral to drive a number of commands into the test pattern generator core.

The command interface maps to the following registers: `cmd_lo` and `cmd_hi`. The command is pushed into the FIFO when the register `cmd_lo` (address 0) is written to. When the FIFO is full, the command

interface asserts the wait request signal. You can create errors by writing to the register `cmd_hi` (address 1). The errors are only cleared when 0 is written to this register or its respective fields. See page "Test Pattern Generator Command Registers" on page 21–12 for more information on the register fields.

Control and Status Interface

The control and status interface is a 32-bit Avalon-MM slave that allows you to enable or disable the data generation as well as set the throttle.

This interface also provides useful generation-time information such as the number of channels and whether or not packets are supported.

Output Interface

The output interface is an Avalon-ST interface that optionally supports packets. You can configure the output interface to suit your requirements.

Depending on the incoming stream of commands, the output data may contain interleaved packet fragments for different channels. To keep track of the current symbol's position within each packet, the test pattern generator core maintains an internal state for each channel.

Instantiating the Test Pattern Generator in SOPC Builder

Use the MegaWizard® interface for the test pattern generator core in SOPC Builder to configure the core. The following sections list the available options in the MegaWizard interface.

Functional Parameter

The functional parameter allows you to configure the test pattern generator as a whole: **Throttle Seed**—The starting value for the throttle control random number generator. Altera recommends a value which is unique to each instance of the test pattern generator and checker cores in a system.

Output Interface

You can configure the output interface of the test pattern generator core using the following parameters:

- **Number of Channels**—The number of channels that the test pattern generator core supports. Valid values are 1–256.
- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1–256. Example—For typical systems that carry 8-bit bytes, set this parameter to 8.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat. Valid values are 1–256.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Error Signal Width (bits)**—The width of the `error` signal on the output interface. Valid values are 0–31. A value of 0 indicates that the `error` signal is not in use.

Test Pattern Checker

This section describes the hardware structure and functionality of the test pattern checker core.

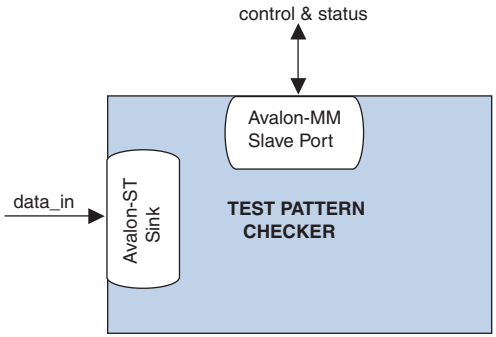
Functional Description

The test pattern checker core accepts data via an Avalon-ST interface, checks it for correctness against the same predetermined pattern used by the test pattern generator core to produce the data, and reports any exceptions to the control interface. You can parameterize most aspects of the test pattern checker's Avalon-ST interface such as the number of error bits and the data signal width, thus allowing you to test components with different interfaces.

The test pattern checker has a throttle register that is set via the Avalon-MM control interface.The value of the throttle register controls the rate at which data is accepted.

Figure 21–2 shows a block diagram of the test pattern checker core.

Figure 21–2. Test Pattern Checker



The test pattern checker core detects exceptions and reports them to the control interface via a 32-element deep internal FIFO. Possible exceptions are data error, missing start-of-packet (SOP), missing end-of-packet (EOP) and signalled error.

As each exception occurs, an exception descriptor is pushed into the FIFO. If the same exception occurs more than once consecutively, only one exception descriptor is pushed into the FIFO. All exceptions are ignored when the FIFO is full. Exception descriptors are deleted from the FIFO after they are read by the control and status interface.

Input Interface

The input interface is an Avalon-ST interface that optionally supports packets. You can configure the input interface to suit your requirements.

Incoming data may contain interleaved packet fragments. To keep track of the current symbol's position, the test pattern checker core maintains an internal state for each channel.

Control and Status Interface

The control and status interface is a 32-bit Avalon-MM slave that allows you to enable or disable data acceptance as well as set the throttle. This interface provides useful generation-time information such as the number of channels and whether the test pattern checker supports packets.

The control and status interface also provides information on the exceptions detected by the test pattern checker core. The interface obtains this information by reading from the exception FIFO.

Instantiating the Test Pattern Checker in SOPC Builder

Use the MegaWizard interface for the test pattern checker core in SOPC Builder to configure the core. The following sections list the available options in the MegaWizard interface.

Functional Parameter

The functional parameter allows you to configure the test pattern checker as a whole: **Throttle Seed**—The starting value for the throttle control random number generator. Altera recommends a unique value to each instance of the test pattern generator and checker cores in a system.

Input Parameters

You can configure the input interface of the test pattern checker core using the following parameters:

- **Number of Channels**—The number of channels that the test pattern checker core supports. Valid values are 1–256.
- **Data Bits Per Symbol**—The number of bits per symbol for the input interface. Valid values are 1–256.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat. Valid values are 1–32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Error Signal Width (bits)**—The width of the `error` signal on the input interface. Valid values are 0–31. A value of 0 indicates that the `error` signal is not in use.

Device and Tools Support

For each device family, the test pattern generator and checker cores provide either full or preliminary support:

- *Full support* means the component meets all functional and timing requirements for the device family and may be used in production designs.
- *Preliminary support* means the component meets all functional requirements, but might still be undergoing timing analysis for the device family; it may be used in production designs with caution.

Figure 21–3 shows the level of support offered by the test pattern generator and checker cores to each Altera device family.

Table 21–3. Device Family Support

Device Family	Support	
	Test Pattern Generator	Test Pattern Checker
Arria™ GX	Preliminary	Preliminary
Cyclone III	Preliminary	Preliminary
Cyclone II	Full	Full
Cyclone	Full	Full
HardCopy® II	Full	Full
Stratix III	Preliminary	Preliminary
Stratix II GX	Full	Full
Stratix II	Full	Full
Stratix GX	Full	Full
Stratix	Full	Full

Installation and Licensing

The test pattern generator and checker cores are included in the Altera MegaCore® IP Library, which is an optional part of the Quartus® II software installation. After you install the MegaCore IP Library, SOPC Builder recognizes these components and can instantiate them into a system.

You can use the test pattern generator and checker for free without a license in any design targeting an Altera device.

Hardware Simulation Considerations

The test pattern generator and checker cores do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

This section describes the software programming model for the test pattern generator and checker cores.

HAL System Library Support

For Nios II processor users, Altera provides HAL system library drivers that enable you to initialize and access the test pattern generator and checker cores. Altera recommends you to use the provided drivers to access the cores instead of accessing the registers directly.

For Nios II IDE users, copy the provided drivers from the following installation folders to your software application directory:

- *<IP installation directory>* **/ip /sopc_builder_ip /altera_avalon_data_source/HAL**
- *<IP installation directory>***/ip/sopc_builder_ip/ altera_avalon_data_sink/HAL**

Software Files

The following software files define the low-level access to the hardware, and provide the routines for the HAL device drivers. Application developers should not modify these files.

- Software files provided with the test pattern generator core:
  - **data_source_regs.h**—The header file that defines the test pattern generator's register maps.
  - **data_source_util.h, data_source_util.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.
- Software files provided with the test pattern checker core:
  - **data_sink_regs.h**—The header file that defines the core's register maps.
  - **data_sink_util.h, data_sink_util.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.

Register Maps

This section describes the register maps for the test pattern generator and checker cores.

Test Pattern Generator Control and Status Registers

Table 21–4 shows the offset for the test pattern generator control and status registers. Each register is 32 bits wide.

Table 21–4. Test Pattern Generator Control and Status Register Map

Offset	Register Name
base + 0	status
base + 1	control
base + 2	fill

Table 21–5 describes the status register bits.

Table 21–5. Status Field Descriptions

Bit(s)	Name	Access	Description
15:0	ID	RO	A constant value of 0x64.
23:16	NUMCHANNELS	RO	The configured number of channels.
30:24	NUMSYMBOLS	RO	The configured number of symbols per beat.
31	SUPPORTPACKETS	RO	A value of 1 indicates packet support.

Table 21–6 describes the control register bits.

Table 21–6. Control Field Descriptions

Bit(s)	Name	Access	Description
0	ENABLE	RW	Setting this bit to 1 enables the test pattern generator core.
7:1	Reserved		
16:8	THROTTLE	RW	Specifies the throttle value which can be between 0 and 256, inclusively. This value is used in conjunction with a pseudorandom number generator to throttle the data generation rate. Setting THROTTLE to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0 and 256 result in a data rate proportional to the throttle value.
17	SOFT RESET	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
31:18	Reserved		

Table 21–7 describes the fill register bits.

Table 21–7. Fill Field Descriptions

Bit(s)	Name	Access	Description
0	BUSY	RO	A value of 1 indicates that data transmission is in progress, or that there is at least one command in the command queue.
6:1	Reserved		
15:7	FILL	RO	The number of commands currently in the command FIFO.
31:16	Reserved		

Test Pattern Generator Command Registers

Table 21–8 shows the offset for the command registers. Each register is 32 bits wide.

Table 21–8. Test Pattern Generator Command Register Map

Offset	Register Name
base + 0	cmd_lo
base + 1	cmd_hi

Table 21–9 describes the cmd_lo register bits. The command is pushed into the FIFO only when the cmd_lo register is written to.

Table 21–9. Cmd_lo Field Descriptions

Bit(s)	Name	Access	Description
15:0	SIZE	RW	The segment size in symbols. Except for the last segment in a packet, the size of all segments must be a multiple of the configured number of symbols per beat. If this condition is not met, the test pattern generator core inserts additional symbols to the segment to ensure the condition is fulfilled.
29:16	CHANNEL	RW	The channel to send the segment on. If the channel signal is less than 14 bits wide, the low order bits of this register are used to drive the signal.
30	SOP	RW	Set this bit to 1 when sending the first segment in a packet. This bit is ignored when packets are not supported.
31	EOP	RW	Set this bit to 1 when sending the last segment in a packet. This bit is ignored when packets are not supported.

Table 21–10 describes the cmd_hi register bits.

Table 21–10. Cmd_hi Field Descriptions

Bit(s)	Name	Access	Description
15:0	SIGNALLED ERROR	RW	Specifies the value to drive the error signal. A non-zero value creates a signalled error.
23:16	DATA ERROR	RW	The output data is XORed with the contents of this register to create data errors. To stop creating data errors, set this register to 0.
24	SUPRESS SOP	RW	Set this bit to 1 to suppress the assertion of the startofpacket signal when the first segment in a packet is sent.
25	SUPRESS EOP	RW	Set this bit to 1 to suppress the assertion of the endofpacket signal when the last segment in a packet is sent.

Test Pattern Checker Control and Status Registers

Table 21–11 shows the offset for the control and status registers. Each register is 32 bits wide.

Table 21–11. Test Pattern Checker Control and Status Register Map

Offset	Register Name
base + 0	status
base + 1	control
base + 2	Reserved
base + 3	
base + 4	
base + 5	exception_descriptor
base + 6	indirect_select
base + 7	indirect_count

Table 21–12 describes the status register bits.

Table 21–12. Status Field Descriptions (Part 1 of 2)

Bit(s)	Name	Access	Description
15:0	ID	RO	Contains a constant value of 0x65.
23:16	NUMCHANNELS	RO	The configured number of channels.

Table 21–12. Status Field Descriptions (Part 2 of 2)

Bit(s)	Name	Access	Description
30:24	NUMSYMBOLS	RO	The configured number of symbols per beat.
31	SUPPORTPACKETS	RO	A value of 1 indicates packet support.

Table 21–13 describes the control register bits.

Table 21–13. Control Field Descriptions

Bit(s)	Name	Access	Description
0	ENABLE	RW	Setting this bit to 1 enables the test pattern checker.
7:1	Reserved		
16:8	THROTTLE	RW	Specifies the throttle value which can be between 0 and 256, inclusively. This value is used in conjunction with a pseudorandom number generator to throttle the data generation rate. Setting THROTTLE to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0 and 256 result in a data rate proportional to the throttle value.
17	SOFT RESET	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
31:18	Reserved		

Table 21–14 describes the exception_descriptor register bits. If there is no exception, reading this register returns 0.

Table 21–14. Exception_descriptor Field Descriptions

Bit(s)	Name	Access	Description
0	DATA ERROR	RO	A value of 1 indicates that an error is detected in the incoming data.
1	MISSINGSOP	RO	A value of 1 indicates missing start-of-packet.
2	MISSINGEOP	RO	A value of 1 indicates missing end-of-packet.
7:3	Reserved		
15:8	SIGNALLED ERROR	RO	The value of the error signal.
23:16	Reserved		
31:24	CHANNEL	RO	The channel on which the exception was detected.

Table 21–15 describes the indirect_select register bits.

Table 21–15. Indirect_select Field Descriptions

Bit	Bits Name	Access	Description
7:0	INDIRECT CHANNEL	RW	Specifies the channel number that applies to the INDIRECT PACKET COUNT, INDIRECT SYMBOL COUNT, and INDIRECT ERROR COUNT registers.
15:8	Reserved		
31:16	INDIRECT ERROR	RO	The number of data errors that occurred on the channel specified by INDIRECT CHANNEL.

Table 21–16 describes the indirect_count register bits.

Table 21–16. Indirect_count Field Descriptions

Bit	Bits Name	Access	Description
15:0	INDIRECT PACKET COUNT	RO	The number of packets received on the channel specified by INDIRECT CHANNEL.
31:16	INDIRECT SYMBOL COUNT	RO	The number of symbols received on the channel specified by INDIRECT CHANNEL.

Test Pattern Generator API

This section describes the application programming interface (API) for the test pattern generator core. All APIs are currently not available from the interrupt service routine (ISR).

data_source_reset()

Prototype: `void data_source_reset(alt_u32 base);`

Thread-safe: No.

Include: <**data_source_util.h**>

Parameters: `base`—The base address of the control and status slave.

Returns: `void`

Description: This function resets the test pattern generator core including all internal counters and FIFOs. The control and status registers are not reset by this function.

data_source_init()

Prototype:	`int data_source_init(alt_u32 base, alt_u32 command_base);`
Thread-safe:	No.
Include:	<**data_source_util.h**>
Parameters:	`base`—The base address of the control and status slave. `command_base`—The base address of the command slave.
Returns:	1—Initialization is successful. 0—Initialization is unsuccessful.
Description:	This function performs the following operations to initialize the test pattern generator core: ● Resets and disables the test pattern generator core. ● Sets the maximum throttle. ● Clears all inserted errors.

data_source_get_id()

Prototype:	`int data_source_get_id(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_source_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The test pattern generator core's identifier.
Description:	This function retrieves the test pattern generator core's identifier.

data_source_get_supports_packets()

Prototype:	`int data_source_init(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_source_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	1—Packets are supported. 0—Packets are not supported.
Description:	This function checks if the test pattern generator core supports packets.

data_source_get_num_channels()

Prototype:	`int data_source_get_num_channels(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_source_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The number of channels supported.
Description:	This function retrieves the number of channels supported by the test pattern generator core.

data_source_get_symbols_per_cycle()

Prototype:	`int data_source_get_symbols(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_source_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The number of symbols transferred in a beat.
Description:	This function retrieves the number of symbols transferred by the test pattern generator core in each beat.

data_source_set_enable()

Prototype:	`void data_source_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	<**data_source_util.h**>
Parameters:	`base`—The base address of the control and status slave. `value`—The `ENABLE` bit is set to the value of this parameter.
Returns:	`void`
Description:	This function enables or disables the test pattern generator core. When disabled, the test pattern generator core stops data transmission but continues to accept commands and stores them in the FIFO.

data_source_get_enable()

Prototype:	`int data_source_get_enable(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_source_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The value of the `ENABLE` bit.
Description:	This function retrieves the value of the `ENABLE` bit.

data_source_set_throttle()

Prototype:	`void data_source_set_throttle(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	<**data_source_util.h**>
Parameters:	`base`—The base address of the control and status slave. `value`—The throttle value.
Returns:	`void`
Description:	This function sets the throttle value, which can be between 0 and 256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern generator sends data.

data_source_get_throttle()

Prototype:	`int data_source_get_throttle(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_source_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The throttle value.
Description:	This function retrieves the current throttle value.

data_source_is_busy()

Prototype:	`int data_source_is_busy(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_source_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	1—The test pattern generator core is busy. 0—The core is not busy.
Description:	This function checks if the test pattern generator is busy. The test pattern generator core is busy when it is sending data or has data in the command FIFO to be sent.

data_source_fill_level()

Prototype:	`int data_source_fill_level(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_source_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The number of commands in the command FIFO.
Description:	This function retrieves the number of commands currently in the command FIFO.

data_source_send_data()

Prototype: `int data_source_send_data(alt_u32 cmd_base, alt_u32 channel, alt_u32 size, alt_u32 flags, alt_u32 error, alt_u32 data_error_mask);`

Thread-safe: No.

Include: <**data_source_util.h**>

Parameters:
`cmd_base`—The base address of the command slave.
`channel`—The channel to send the data on.
`size`—The data size.
`flags`—Specifies whether to send or suppress SOP and EOP signals. Valid values are `DATA_SOURCE_SEND_SOP`, `DATA_SOURCE_SEND_EOP`, `DATA_SOURCE_SEND_SUPRESS_SOP` and `DATA_SOURCE_SEND_SUPRESS_EOP`.
`error`—The value asserted on the `error` signal on the output interface.
`data_error_mask`—This parameter and the data are XORed together to produce erroneous data.

Returns: Always returns 1.

Description: This function sends a data fragment to the specified channel.

If packets are supported, user applications must ensure the following conditions are met:

- SOP and EOP are used consistently in each channel.
- Except for the last segment in a packet, the length of each segment is a multiple of the data width.

If packets are not supported, user applications must ensure the following conditions are met:

- No SOP and EOP indicators in the data.
- The length of each segment in a packet is a multiple of the data width.

Test Pattern Checker API

This section describes the API for the test pattern checker core. The APIs are currently not available from the ISR.

data_sink_reset()

Prototype:	`void data_sink_reset(alt_u32 base);`
Thread-safe:	No.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	`void`
Description:	This function resets the test pattern checker core including all internal counters.

data_sink_init()

Prototype:	`int data_source_init(alt_u32 base);`
Thread-safe:	No.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	1—Initialization is successful. 0—Initialization is unsuccessful.
Description:	This function performs the following operations to initialize the test pattern checker core: • Resets and disables the test pattern checker core. • Sets the throttle to the maximum value.

data_sink_get_id()

Prototype:	`int data_sink_get_id(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The test pattern checker core's identifier.
Description:	This function retrieves the test pattern checker core's identifier.

data_sink_get_supports_packets()

Prototype:	`int data_sink_init(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	1—Packets are supported. 0—Packets are not supported.
Description:	This function checks if the test pattern checker core supports packets.

data_sink_get_num_channels()

Prototype:	`int data_sink_get_num_channels(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The number of channels supported.
Description:	This function retrieves the number of channels supported by the test pattern checker core.

data_sink_get_symbols_per_cycle()

Prototype:	`int data_sink_get_symbols(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The number of symbols received in a beat.
Description:	This function retrieves the number of symbols received by the test pattern checker core in each beat.

data_sink_set enable()

Prototype:	`void data_sink_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave. `value`—The `ENABLE` bit is set to the value of this parameter.
Returns:	`void`
Description:	This function enables the test pattern checker core.

data_sink_get_enable()

Prototype:	`int data_sink_get_enable(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The value of the `ENABLE` bit.
Description:	This function retrieves the value of the `ENABLE` bit.

data_sink_set_throttle()

Prototype:	`void data_sink_set_throttle(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave. `value`—The throttle value.
Returns:	`void`
Description:	This function sets the throttle value, which can be between 0 and 256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern checker receives data.

data_sink_get_throttle()

Prototype:	`int data_sink_get_throttle(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The throttle value.
Description:	This function retrieves the throttle value.

data_sink_get_packet_count()

Prototype:	`int data_sink_get_packet_count(alt_u32 base, alt_u32 channel);`
Thread-safe:	No.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave. `channel`—Channel number.
Returns:	The number of packets received on the given channel.
Description:	This function retrieves the number of packets received on a given channel.

data_sink_get_symbol_count()

Prototype:	`int data_sink_get_symbol_count(alt_u32 base, alt_u32 channel);`
Thread-safe:	No.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave. `channel`—Channel number.
Returns:	The number of symbols received on the given channel.
Description:	This function retrieves the number of symbols received on a given channel.

data_sink_get_error_count()

Prototype:	`int data_sink_get_error_count(alt_u32 base, alt_u32 channel);`
Thread-safe:	No.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave. `channel`—Channel number.
Returns:	The number of errors received on the given channel.
Description:	This function retrieves the number of errors received on a given channel.

data_sink_get_exception()

Prototype:	`int data_sink_get_exception(alt_u32 base);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status slave.
Returns:	The first exception descriptor in the exception FIFO. 0—No exception descriptor found in the exception FIFO.
Description:	This function retrieves the first exception descriptor in the exception FIFO and pops it off the FIFO.

data_sink_exception_is_exception()

Prototype:	`int data_sink_exception_is_exception(int exception);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`exception`—Exception descriptor
Returns:	1—Indicates an exception. 0—No exception.
Description:	This function checks if a given exception descriptor describes a valid exception.

data_sink_exception_has_data_error()

Prototype:	`int data_sink_exception_has_data_error(int exception);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`exception`—Exception descriptor
Returns:	1—Data has errors. 0—No errors.
Description:	This function checks if a given exception indicates erroneous data.

data_sink_exception_has_missing_sop()

Prototype:	`int data_sink_exception_has_missing_sop(int exception);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`exception`—Exception descriptor.
Returns:	1—Missing SOP. 0—Other exception types.
Description:	This function checks if a given exception descriptor indicates missing SOP.

data_sink_exception_has_missing_eop()

Prototype:	`int data_sink_exception_has_missing_eop(int exception);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`exception`—Exception descriptor.
Returns:	1—Missing EOP. 0—Other exception types.
Description:	This function checks if a given exception descriptor indicates missing EOP.

data_sink_exception_signalled_error()

Prototype:	`int data_sink_exception_signalled_error(int exception);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`exception`—Exception descriptor.
Returns:	The signalled error value.
Description:	This function retrieves the value of the signalled error from the exception.

data_sink_exception_channel()

Prototype:	`int data_sink_exception_channel(int exception);`
Thread-safe:	Yes.
Include:	<**data_sink_util.h**>
Parameters:	`exception`—Exception descriptor.
Returns:	The channel number on which the given exception occurred.
Description:	This function retrieves the channel number on which a given exception occurred.

Referenced Document

This chapter references the *Avalon Streaming Interface Specification*.

Document Revision History

Table 21–17 shows the revision history for this chapter.

Table 21–17. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Initial release.	—